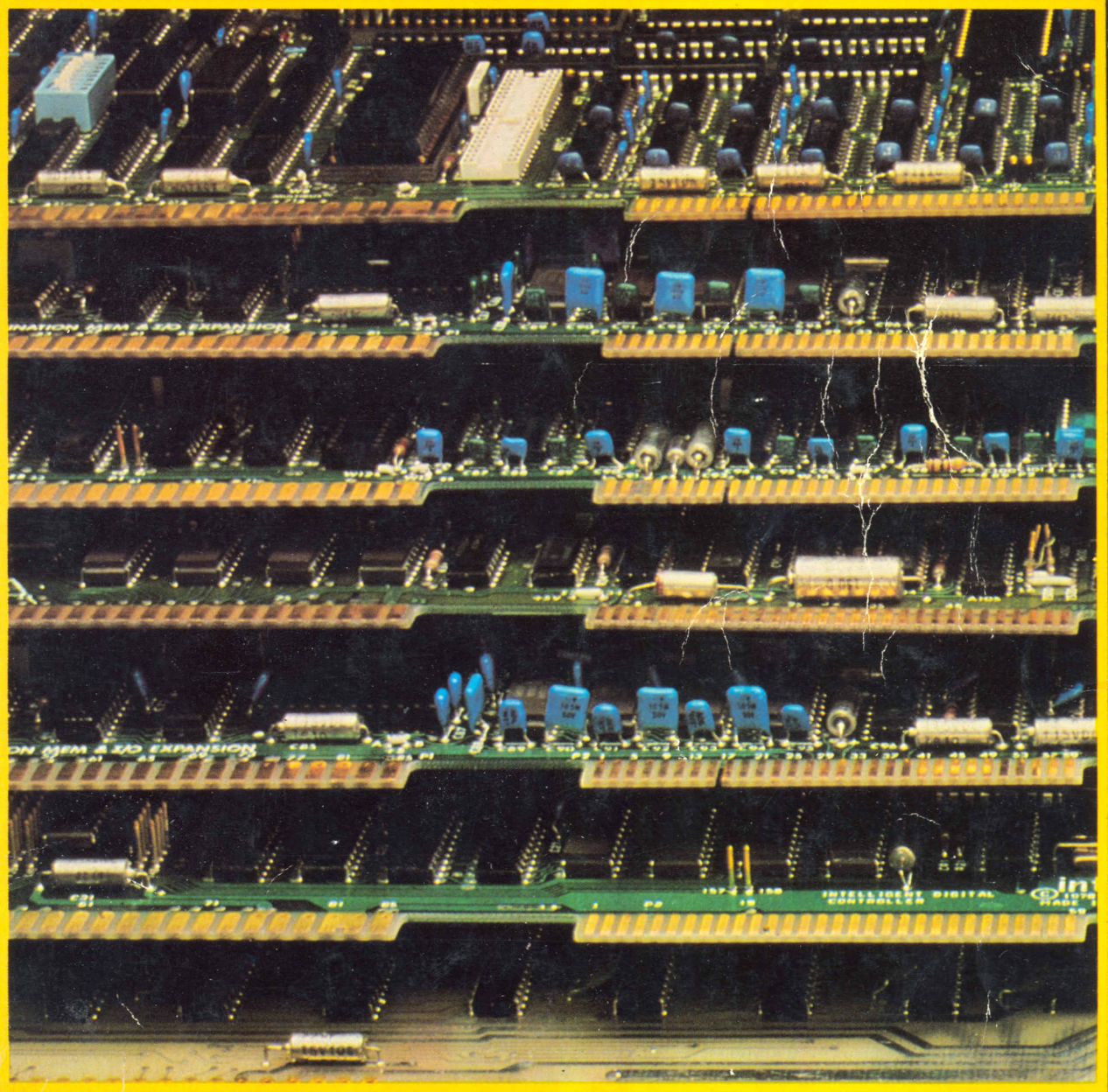


intel®

iSBC Applications Handbook



intel®



quantum electronics

Box 391262

Bramley

2018

iSBC™ APPLICATIONS HANDBOOK

SEPTEMBER 1981

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or distribution without the prior written consent of Intel Corporation is prohibited. Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuitry or patent licenses are implied.

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and may only be used to identify Intel products:

BXP	Intelvision	MULTIBUS
CREDIT	Intellic	MULTIMODULE
I	iSBC	Plug-A-Bubble
ICE	iSBX	PROMPT
ICS	Library Manager	Program
Im	MCS	RMK
Intel	Megachassis	UPI
	Microminitronics	Ascon
	Micromap	System 3000

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Motorola Data Sciences Corporation.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Department SV-3
3065 Bowers Avenue
Santa Clara, CA 95051

Intel

multibus electronics

Box 341545

Volume 8

2010



ISBC™ APPLICATIONS HANDBOOK

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9 (a) (9). Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and may only be used to identify Intel products:

BXP	Inteleview	MULTIBUS
CREDIT	Inteltec	MULTIMODULE
i	iSBC	Plug-A-Bubble
ICE	iSBX	PROMPT
ICS	Library Manager	Promware
i _m	MCS	RMX
Insite	Megachassis	UPI
Intel	Micromainframe	μScope
	Micromap	System 2000

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Department SV3-3
3065 Bowers Avenue
Santa Clara, CA 95051

PREFACE

Since Intel introduced the iSBC 80/10 Single Board Computer in early 1976, the family of Intel OEM Microcomputer Systems has grown rapidly. Original equipment manufacturers and volume end-users alike have responded to the concept originated by Intel of having all the functions of a computer — central processing unit, memory, input-output and system expansion capability — present on one printed circuit board.

The capabilities of a single board computer have been enhanced by the creation of the industry-standard MULTIBUS system bus. System expansion boards have been introduced for memory, serial I/O and parallel I/O expansion, as well as analog I/O, DMA controllers and peripheral controllers. A unique feature of the MULTIBUS architecture, however, is its capability to support multiple single board computers. This capability permits sophisticated multiprocessing configurations using standard off-the-shelf 8-bit and 16-bit single board computers. The introduction of the iSBX MULTIMODULE expansion boards has revolutionized the concept of the single board computer. Now iSBC host boards may be custom configured with iSBX expansion boards based on the I/O requirements of the application. This capability provides lower cost, higher performance single board solutions. Powerful software tools like the iRMX 80 and iRMX 88 Real-Time Executives and the iRMX 86 Operating System are key members of the iSBC product family. They provide users with the tools for quick implementations of simple or complex systems. iCS product line provides chassis and signal conditioning/termination strips as well as board level products which were developed specifically for industrial users.

This application manual is divided into three sections: iSBC Hardware, iSBC Software and iCS Products. It contains all of the current application notes, reliability reports, magazine articles and professional journal reprints on the products of the Intel iSBC product family. We have compiled all of this information into a single publication for your convenience. Please contact us with your questions, comments, and suggestions on how we may provide you with useful information on our products.

INTEL CORPORATION
OEM Microcomputer Systems
Applications Engineering
Hillsboro, Oregon 97123

AR-88	Table B: Applications of a Single Board Computer	1-221
AR-89	Design Time by Using Single Board Computer Applications	1-221
AR-90	Reduce Your iC-Based System	1-221
AR-91	Controller Board	1-229
AR-92	16-Bit Single Board Computer	1-237
AR-93	Maintains 8-Bit Family Ties	1-237
AR-94	A New Family of MULTIMODULE Boards Expands the Solutions Provided by Intel's Single Board Computers	1-245
AR-95	Special Function Modules Ride on Computer Board	1-249
AR-96	Multiprocessing System Mixes 8- and 16-Bit Microcomputers	1-257
AR-97	Using FORTRAN-80 for iSBC Applications	2-33
AR-98	RMX80 Real-Time Multitasking Executive	2-3
AR-99	Using FORTRAN-80 for iSBC Applications	2-33
AR-100	Designing and Assembling Microcomputer Systems Grows Easier	3-123
AR-101	iSBC 8080/85 Intelligent Digital Processors	3-61
AR-102	Closed Loop Control Using the iSBC 8080/85 Intelligent Digital Processors	3-61
AR-103	Series in Control Applications	3-3
AR-104	Using Intel's Industrial Control Series in Control Applications	3-3
AR-105	Multitasking Executive Speeds 16-Bit Micro	2-329
AR-106	Applications	2-321
AR-107	Foundation for Microprocessor	2-321
AR-108	A Small-Scale Operating System	2-315
AR-109	Cuts Cost of 16-Bit OS Design	2-315
AR-110	Technical Literature List	4-5
AR-111	Related Intel Publications	4-3
AR-112	Documentation	4-3

ISBC HARDWARE

AP-26	iSBC 80/10A-SYSTEM 80/10 Single Board Computer Applications	1-3
AP-28A	Intel MULTIBUS Interfacing	1-45
AP-43	Using the iSBC 957 Execution Vehicle for Executing 8086 Program Code	1-79
AP-53	Using the iSBC 544 Intelligent Communications Controller	1-111
AP-96	Designing iSBX MULTIMODULE Boards	1-175
AR-48	Reduce Your μ C-Based System Design Time by Using Single Board Microcomputers	1-199
AR-55	Design Motivations for Multiple Processor Microcomputer Systems	1-211
AR-65	Triple-Bus Architecture on a Single Board Microcomputer ...	1-221
AR-69	Dual-Port RAM Hikes Throughput in Input/Output Controller Board	1-229
AR-72	16-Bit Single Board Computer Maintains 8-Bit Family Ties	1-237
AR-122	A New Family of MULTIMODULE Boards Extends the Solutions Provided by Intel's Single Board Computers	1-245
AR-123	Special Function Modules Ride on Computer Board	1-249
AR-133	Multiprocessing System Mixes 8- and 16-Bit Microcomputers ...	1-257

ISBC SOFTWARE

AP-33	RMX/80 Real-Time Multitasking Executive	2-3
AP-47	Using FORTRAN-80 for iSBC Applications	2-33

ISBC SOFTWARE (con't.)

AP-86	Using the iRMX 86 Operating System	2-73
AP-88	Multiprocessing Extensions for the RMX/80 Real-Time Executive	2-131
AP-109	Using Intel Single Board Computers for Serial Distributed Processing Links	2-173
AP-110	Using the iRMX 86 Operating System on iAPX 86 Component Designs	2-243
AR-41	An Integral Real-Time Executive for Microcomputers	2-303
AR-124	Introducing the RMX/86 Real-Time, Multitasking, 16-Bit Operating System	2-311
AR-125	Modular Multitasking Executive Cuts Cost of 16-Bit OS Design ..	2-315
	A Small-Scale Operating System Foundation for Microprocessor Applications	2-321
AR-172	Multitasking Executive Speeds 16-Bit Micros	2-329

ICS PRODUCTS

AP-52	Using Intel's Industrial Control Series in Control Applications ..	3-3
AP-60	Closed Loop Control Using the iSBC 569/941 Intelligent Digital Processors	3-61
AR-91	Designing and Assembling Microcomputer Systems Grows Easier	3-123

DOCUMENTATION

Related Intel Publications	4-3
Technical Literature List	4-5

iSBC™ Hardware

1

1

i2BC™ Hardware



APPLICATION NOTE

AP-26

1-1	INTRODUCTION
1-2	OVERVIEW
1-3	SBC CONFIGURATION OPTIONS
1-4	Serial I/O Options
1-5	Parallel I/O Options
1-6	Bus Interfacing
1-7	APPLICATIONS
1-8	Instrumentation
1-9	Communication
1-10	Process Control
1-11	I/O Device Controller
1-12	CONCLUSION
1-13	APPENDIX A — ISBC SW10A
1-14	SCHEMATICS

iSBC 80/10A — SYSTEM 80/10 Single Board Computer Applications

Thomas Rolander
Microcomputer Applications

iSBC 80/10A-SYSTEM 80/10 Single Board Computer Applications

Contents

INTRODUCTION	1-5
OVERVIEW	1-5
SBC CONFIGURATION OPTIONS	1-7
Serial I/O Options	1-7
Parallel I/O Options	1-8
Bus Interfacing	1-8
APPLICATIONS	1-10
Instrumentation	1-10
Communication	1-15
Process Control	1-23
I/O Device Controller	1-27
CONCLUSION	1-31
APPENDIX A — iSBC 80/10A SCHEMATICS	1-33

INTRODUCTION

The recent entry of the single board computer into the broad field of electronic applications is substantiating the billing as a "super component". Single board computers provide a solution to several problems that have not been solved by the use of conventional computers: cost, size, and design specialization.

Many potential microcomputer applications have been overlooked because of the design tasks required to build a microcomputer system. These tasks traditionally include interfacing of the system clock, read/write memory, I/O ports and drivers, serial communications interface, bus control logic and drivers. Intel's iSBC 80/10A enables the design engineer to concentrate on the application of microcomputers, rather than on implementation details.

This application note begins with an overview of the Intel® iSBC 80/10A. Readers who are familiar with the iSBC 80/10A may choose to skip to the applications section, which describes the following typical iSBC 80/10A applications:

- The iSBC 80/10A used for instrumentation control of a Fluke 8375 Digital Multimeter.
- The iSBC 80/10A used as a SCADA Terminal in a communication application.
- The iSBC 80/10A used for temperature monitoring in a process control application.
- The iSBC 80/10A used as an interrupt driven device controller for a Centronics printer.

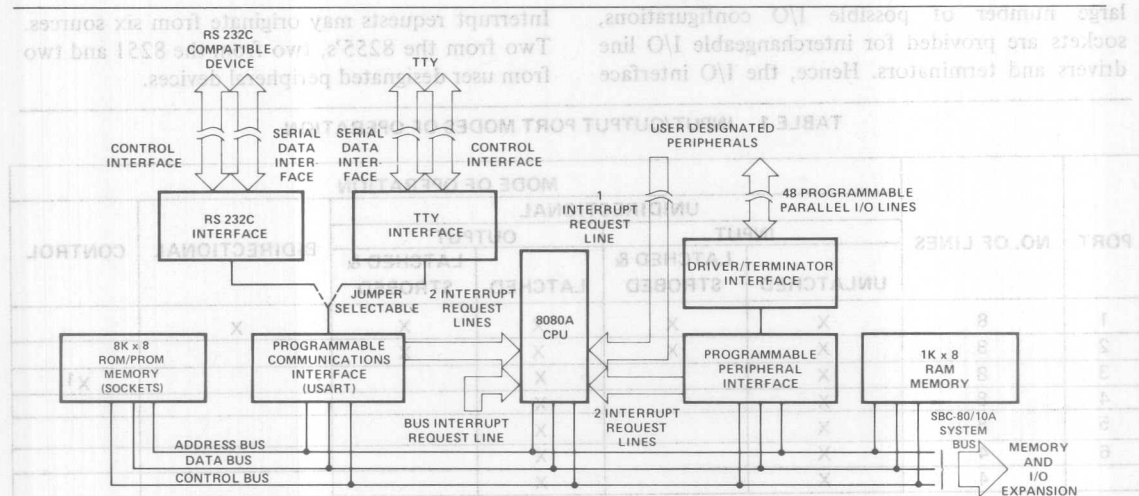
Each example shows the user program and hardware required for the application. The program listings are interspersed with the text describing the application. Both 8080 Macro Assembly Language and Intel's PL/M-80 are used in the examples.

The software was developed on an Intel® Microcomputer Development System (MDS). The MDS provided the tools necessary to edit, assemble or compile, link and locate the application software. Hardware development was facilitated by the use of Intel's In-Circuit Emulator (ICE 80). For further information regarding the Microcomputer Development System, the reader is referred to the publications listed at the beginning of this application note.

OVERVIEW

The iSBC 80/10A is a member of Intel's complete line of OEM computer systems which take full advantage of Intel's LSI technology to provide economical, self-contained computer based solutions for OEM applications. The iSBC 80/10A is a complete computer system on a single 6.75-by-12 inch printed circuit card. A block diagram of the iSBC 80/10A is shown in Figure 1.

Intel's powerful 8-bit n-channel MOS 8080A CPU, fabricated on a single LSI chip, is the central processor for the iSBC 80/10A. The 8080A contains six 8-bit general purpose registers and an accumulator. The six general purpose registers may be addressed individually or in pairs, providing both single and double precision operators.



1. Interrupts originating from the Programmable Communications Interface and Programmable Peripheral Interface are jumper selectable.

Figure 1. iSBC 80/10A Block Diagram

The 8080A has a 16-bit program counter which allows direct addressing of up to 64K bytes of memory. An external stack, located within any portion of read/write memory, may be used as a last in/first out stack to store the contents of the program counter, flags, accumulator and all of the six general purpose registers. A 16-bit stack pointer addresses the external stack. This provides sub-routine nesting that is bounded only by memory size.

The iSBC 80/10A contains 1K bytes of read/write memory using Intel's low power static RAM. All on board RAM read and write operations are performed at maximum processor speed. Four sockets for up to 8K bytes of non-volatile read-only memory are provided on the board. Read-only memory may be added in 1K byte increments (up to 4K total) using Intel® 8708 erasable and electrically reprogrammable ROMs (EPROMs) or Intel 8308 masked ROMs. Optionally, if more than 4K bytes are required, read only memory may be added in 2K byte increments (up to 8K total) using Intel® 2716 EPROMs or 2316E masked ROMs. All on-board ROM or EPROM read operations are performed at maximum processor speed.

The iSBC 80/10A contains 48 programmable parallel I/O lines implemented using two Intel® 8255 Programmable Peripheral Interfaces. The system software is used to configure the I/O lines in any combination of unidirectional input/output, and bidirectional ports indicated in Table I. Therefore, the I/O interface may be customized to meet specific peripheral requirements. To support the large number of possible I/O configurations, sockets are provided for interchangeable I/O line drivers and terminators. Hence, the I/O interface

provides the appropriate combination of optional line drivers and terminators to allow the required sink current, polarity, and drive/termination characteristics for each application. The 48 programmable I/O lines and signal ground lines are brought out to two 50-pin edge connectors that mate with flat, round, or woven cable.

A programmable communications interface using Intel's 8251 Universal Synchronous/Asynchronous Receiver/Transmitter (USART) is contained on the iSBC 80/10A. A jumper selectable baud rate generator provides the 8251 with all common communication frequencies. The 8251 can be programmed by the user's system software to select the desired asynchronous or synchronous serial data transmission technique (including IBM Bi-sync). The mode of operation (synchronous or asynchronous), data format, control character format, parity, and asynchronous transmission rate are all under program control. The 8251 provides full duplex, double buffered transmission and receive capability. Parity, overrun, and framing error detection circuits are all incorporated in the 8251. The inclusion of jumper selectable TTY or EIA RS232C compatible interfaces on the board, in conjunction with the 8251, provide a direct interface to teletypes, CRTs, asynchronous and synchronous modems, and other RS232C compatible devices. The RS232C or TTY command lines, serial data lines, and signal ground lines are brought out to a 25-pin edge connector that mates with RS232C compatible flat, round, or woven cable.

Interrupt requests may originate from six sources. Two from the 8255's, two from the 8251 and two from user designated peripheral devices.

TABLE 1 INPUT/OUTPUT PORT MODES OF OPERATION

PORT	NO. OF LINES	MODE OF OPERATION					
		UNIDIRECTIONAL				BIDIRECTIONAL	CONTROL
		INPUT		OUTPUT			
		UNLATCHED	LATCHED & STROBED	LATCHED	LATCHED & STROBED		
1	8	X	X	X	X	X	
2	8	X	X	X	X		
3	8	X		X			x1
4	8	X		X			
5	8	X		X			
6	4	X		X			
	4	X		X			

1. Note: Port 3 must be used as a control port when either Port 1 or Port 2 are used as a latched and strobed input or a latched and strobed output or Port 1 is used as a bidirectional port.

The 8255's can generate interrupts when a byte of information is ready to be transferred to the CPU (i.e., input buffer full) or a byte of information has been transferred to a peripheral device (i.e., output buffer is empty).

The 8251 can generate interrupts when a character is ready to be transferred to the CPU (i.e., receive channel buffer is full) or a character is ready to be transmitted (i.e., transmit channel data buffer is empty).

The user designated peripheral devices can generate two interrupts: one via the system bus and the other via the I/O edge connector.

The two interrupts from the 8255's and the two interrupts from the 8251 are all individually maskable under program control. The six interrupt request lines share a single CPU interrupt level. When an interrupt request is recognized, a RESTART 7 instruction is generated. The processor responds by suspending program execution and making a subroutine call to a user defined interrupt service routine originating at location 38 (Hexadecimal).

iSBC 80/10A memory and I/O capacity may be increased by adding standard Intel memory and I/O boards. Modular expandable backplanes and card cages, each with a four-board capacity, are available to support multi-board systems.

The development cycle of iSBC 80/10A based products may be significantly reduced using the Intel Microcomputer Development System. The resident macro-assembler, PL/M-80 compiler, text editor, and system monitor greatly simplify the design, development, and debug of user designed iSBC 80/10A system software. A diskette-based system allows programs to be loaded, assembled, edited, and executed faster than using conventional paper tape, card, or cassette peripherals. A unique In-Circuit Emulator (ICE 80) provides the capability of developing and debugging software directly on the iSBC 80/10A.

iSBC CONFIGURATION OPTIONS

The iSBC 80/10 provides the user with a powerful and flexible I/O capability for both parallel and serial transfers. This section discusses the user programmable and jumper-selectable options, and bus interfacing.

SERIAL I/O OPTIONS

The serial I/O interface, using Intel's 8251 USART, provides a serial data communications channel that can be programmed to operate with most of the

current serial data transmission protocols. There are three general areas of serial I/O options:

1. choice of interface type, RS232C or current loop,
2. baud rate and program-selectable mode options,
3. choice of an interrupt mechanism.

The user has the choice, through jumper connections, of configuring the serial I/O logic to present either an RS232C or a 20 mA current loop interface to an external device. If an RS232C interface is used, the 8251 can assume the role of a "data set" or a "data processing terminal". This enables the serial interface to be connected to different devices such as modems and computer terminals.

There are two factors which enter into the choice of baud rate. They are the actual clock frequency used to drive the transmit/receive clocks on the 8251 and the baud rate factor selected by a programmable mode instruction control word output by the processor to the 8251. The baud rate factor is used to effectively divide the 8251 transmit and receive clocks by 1, 16 or 64. During normal operation a factor of 16 is selected for asynchronous transmissions from 9.6K to 300 baud. A factor of 64 must be used to achieve a baud rate of 110. The baud rate factor is only applicable to asynchronous transmission, as all synchronous transmission is done with an implied factor of one.

Before beginning serial I/O operations, the 8251 must be program-initialized to support the desired mode of operation. The CPU initializes the 8251 by issuing a set of control bytes to the USART device. These control words specify:

- synchronous or asynchronous operation
- baud rate factor
- character length
- number of stop bits
- even/odd parity
- parity/no parity

Refer to the *iSBC 80/10 and iSBC 80/10A Single Board Computer Hardware Reference Manual* or the "8251 Application Note" for details on the control words used to direct the operation of the 8251.

The serial I/O logic can be configured with different forms of interrupt request mechanisms. By connecting a jumper, the user can allow the 8251's Receiver Ready output to generate an interrupt request. The Receiver Ready output goes high whenever the Receiver Enable bit of the command

word has been set and the 8251 contains a character that is ready to be input to the CPU. The user can also choose to have the 8251's Transmitter Ready or Transmitter Empty output activate the interrupt request. The Transmitter Empty goes high when the 8251 has no characters to transmit. Transmitter Ready is high when the 8251 is ready to accept a character from the CPU. Both Transmitter Empty and Transmitter Ready are enabled by setting the Transmit Enable bit of the command word. Upon receiving an interrupt, the program can determine the actual condition which is responsible for the interrupt by reading the status of the 8251 device.

PARALLEL I/O OPTIONS

The parallel I/O interface consists of six 8-bit I/O ports implemented with two Intel 8255 Programmable Peripheral Interface devices. Eight lines already have a bidirectional driver and termination network permanently installed. The remaining 40 lines are uncommitted. Sockets are provided for the installation of active driver networks or passive termination networks as required to meet the specific needs of the user system.

The primary considerations in determining how to use each of the six I/O ports are:

1. choice of operating mode,
2. direction of data flow (input, output or bidirectional),
3. selection of interrupt mechanism,
4. choice of driver/termination networks for the port's data path.

Operating Modes. There are three basic operating modes that can be selected by the system software. The modes of operation will be described here in general terms, leaving it to the reader to obtain details from the *iSBC 80/10 and iSBC 80/10A Single Board Computer Hardware Reference Manual* or the "8255 Application Note."

Mode 0 is a basic input/output functional configuration which provides simple input and output operations. No "handshaking" is required, data is simply written to or read from a specified port. The outputs are latched and the inputs are unlatched.

Mode 1 is a strobed input/output functional configuration which provides a means for transferring I/O data to or from a specified port in conjunction with strobes or handshaking signals. The outputs are latched and are accompanied by

an output control line which indicates that the processor has loaded the output port with a data byte. The input data is latched when accompanied by its externally operated control signal.

Mode 2 is a strobed bidirectional bus input/output functional configuration which provides a means for communicating with a peripheral device or structure on a single 8-bit bus for both transmitting and receiving data. Handshaking signals are provided to maintain proper bus flow discipline in a manner similar to mode 1.

Data Flow Direction. In addition to the choice of operating mode, the user may also specify the direction of data flow, input or output from the 8255's. At the time of RESET, the 8255's are configured into the input mode until altered by a control word directed to the control word register. When an output mode control word is received, all of the data bits are set to the low output state.

Interrupt Mechanism. When the 8255 is programmed to operate in mode 1 or mode 2, control signals are provided that can be used as interrupt request inputs to the CPU. The interrupt request signals, generated from one of the ports (port C), can be inhibited or enabled by setting or resetting the associated interrupt enable flip-flop, using the bit set/reset function of port C. This function allows the programmer to mask the interrupts from specific I/O devices without affecting any other device in the interrupt structure.

Driver/Termination Networks. Depending on the direction of data flow, the user will select the appropriate TTL line drivers and Intel terminators that are compatible with the I/O driver/terminator sockets on the iSBC 80/10A. The list of suitable line drivers includes those with inverting, non-inverting, and open collector characteristics. There are two types of terminators: a 220-ohm/330-ohm divider or a 1K ohm pull-up.

BUS INTERFACING

The system bus interface logic consists of three general groups of circuitry:

1. gates that accept the various bus control signals, the interrupt request lines, and the ready indications, and then apply these signals to the CPU logic elements,
2. the system bus drivers,
3. the failsafe circuitry which generates an acknowledgment during interrupt sequences and during those cycles in which an ac-

knowledge is not returned because a non-existent device was inadvertently addressed.

Bus Interface Signals. The following paragraphs describe portions of the system bus interfacing logic relevant to interfacing a user device to the iSBC 80/10A. (Note: Whenever a signal is active-low, its mnemonic is followed by a slash; for example, MRDC/ means that the level on that line will be low when the memory read command is true.)

BCLK/ — Bus clock; used to synchronize bus control circuits on all master modules. BCLK/ has a frequency of 9.216 MHz. BCLK/ may be slowed, stopped or single stepped, if desired.

INIT/ — Initialization signal; resets the entire system to a known internal state.

BPRN — Bus priority input signal; indicates to the iSBC 80/10A that a higher priority master module is requesting use of the system bus. BPRN suspends the processing activity and drivers of the iSBC 80/10A until the signal goes low.

BUSY/ — Bus busy signal; indicates that the bus is currently in use. BUSY/ prevents all other master modules from gaining control of the bus. BUSY/ is driven by the HLDA/ output from the iSBC 80/10A in response to a BPRN input. It indicates that the bus is available.

MRDC/ — Memory read command; indicates that the address of a memory location has been placed on the system address lines and specifies that the contents of the addressed location are to be read and placed on the system data bus.

MWTC/ — Memory write command; indicates that the address of a memory location has been placed on the system address lines and that a data word has been placed on the system data bus. MWTC/ specifies that the data word is to be written into the addressed memory location.

IORC/ — I/O read command; indicates that the address of an input port has been placed on the system address bus and that the data at that input port is to be read and placed on the system data bus.

IOWC/ — I/O write command; indicates that the address of an output port has been placed on the system address bus and that the contents

of the system data bus are to be output to the addressed port.

XACK/ — Transfer acknowledge signal; the required response of an external memory location or I/O port which indicates that the specified read/write operation has been completed (that is, data has been placed on, or accepted from, the system data bus lines).

AACK/ — An advance acknowledge, in response to a memory read or write command, that allows the memory to complete the specified operation without requiring the CPU to wait.

CCLK/ — Constant clock; provides a clock signal of constant frequency (9.216 MHz) for use by optional memory and I/O expansion boards. The same signal is used to drive both CCLK/ and BCLK/.

INTR1/ — Externally generated interrupt request.

ADRO/—ADRF/ — 16 Address lines; used to transmit the address of the memory location or I/O port to be accessed. ADRF/ is the most significant bit.

DAT0/—DAT7/ — Bidirectional data lines; used to transmit/receive information to/from a memory location or I/O port. DAT7/ is the most significant bit.

Bus Acknowledges. Further distinction between transfer acknowledge (XACK/) and advance acknowledge (AACK/) is required. All external memory and I/O transfer requests must return XACK/ to the iSBC 80/10A (even if AACK/ is also returned). XACK/ indicates that data has been placed on (read command) or accepted from (write command) the system data bus lines. AACK/ is an advance acknowledge in response to a memory or I/O port command. It has been provided because the 8080A samples the ready line before valid data is required on the bus. If this condition is properly anticipated, AACK/ can be returned before the data is actually read, thus allowing an earlier operation to be completed. AACK/ should be used only with a thorough understanding of the additional information provided in the *iSBC 80/10 and iSBC 80/10A Single Board Computer Hardware Reference Manual. DMA Transfers*. An external device can make DMA transfers to or from RAM expansion boards. The transfer is coordinated with the iSBC 80/10A by means of two bus signals: bus priority input (BPRN) and bus busy (BUSY/). The first step in making a DMA transfer is to obtain control of the system bus. This is

achieved by asserting BRPN to the iSBC 80/10A and then waiting until the iSBC 80/10A returns BUSY/, indicating that it has relinquished control of the system bus. When this step is completed the external device may proceed with its DMA transfers until it is finished. At that time BPRN should be removed to allow the iSBC 80/10A to regain control of the system bus. It should be noted that the iSBC 80/10A is placed in a hold state when it does not have control of the system bus.

APPLICATIONS

The iSBC 80/10A may be applied to a wide variety of applications. Specific applications in four areas are presented in this application note. They are presented to illustrate a broad spectrum of single board computer capabilities and to demonstrate the use of various system features.

INSTRUMENTATION

Microprocessors have been used in instrumentation for many tasks ranging from handling simple interface functions to control of the analog to digital conversion process. The use of a single board computer can further serve in the application of instruments themselves to laboratory or process control environments. It is quite often necessary in these applications to control instrumentation remotely. A number of rather expensive minicomputer-controlled solutions now exist on the market as automatic test equipment (ATE) systems. The iSBC 80/10A presents itself as a cost effective solution in situations where the larger ATE systems are beyond economic justification.

The iSBC 80/10A can be the sole CPU element in the system, providing instrumentation control and computational capability; or it can supplement a larger host CPU by handling distributed processing requirements.

Instrumentation Control Application Example

Most instruments such as DVMs, counters, data loggers, synthesizers, etc., have optional data output units (DOUs) and/or remote control units (RCUs). It is particularly time consuming to interface each instrument's DOU/RCU with custom-digital logic. Until the recent IEEE-488 interface standard, there was little in common from one interface to the next. The parallel I/O lines of the iSBC 80/10A provide a common interface element that can be adapted to a majority of the DOUs and RCUs available today by means of software.

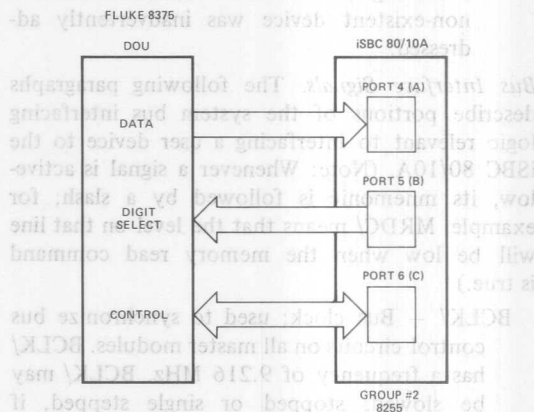


Figure 2. Interface Block Diagram

This instrumentation control application shows how the iSBC 80/10A has been used to control and read the data from the data output unit (DOU) of a Fluke 8375 Digital Multimeter.

Interfacing the iSBC 80/10A to the Fluke 8375 DOU has been accomplished through the use of three parallel I/O ports shown in Figure 2. An 8-bit port has been used to read input data from the Fluke 8375 DOU. Another 8-bit port has been used to control the multiplexing of data from the DOU to the iSBC 80/10A. And, an 8-bit port has been used to provide the required control and monitoring of the following DOU functions: busy flag, sample sync flag, timeout enable, external trigger and trigger inhibit.

The following listing contains a complete program to provide the necessary interface control functions as well as an exercise program. The program listing is interspersed with text that is used to clarify the elements of the program.

```

0  ; INSTRUMENTATION CONTROL APPLICATION
1  ;
2  ; FLUKE 8375 DIGITAL MULTIMETER
3  ;
4  ; DATA OUTPUT UNIT (DOU) CONTROLLER
5  ;
6  ;
7  ;
8  ;

```

The CSEG directs the ISIS-II 8080 Assembler to generate a relocatable code segment. Relocatable code can later be placed at any memory address by Intel's LOCATE program. This lets you write your program without worrying about the application's final memory configuration.

```

9  ;
10 ; CSEG
11 ;
12 ;
13 ;

```

Equate Table. The following table is used to give symbolic names to the binary I/O port addresses. The names used later in the program increase readability.

```

14 ;
15 ;
16 EQUATE TABLE
17 ;
18 CWR EQU 0E8H ; 8255 #2 CONTROL WORD REGISTER
19 DATIN EQU 0E8H ; DECADE PAIR DATA INPUT PORT
20 STB EQU 0E9H ; STROBE OUTPUT PORT
21 FLG EQU 0EAH ; FLAG INPUT PORT
22 TRG EQU 0EAH ; TRIGGER OUTPUT PORT
23 ;
24

```

The exercise program uses some of the subroutines provided in the iSBC 80/10A System Monitor PROMs. The addresses of the subroutines are included in the equate table.

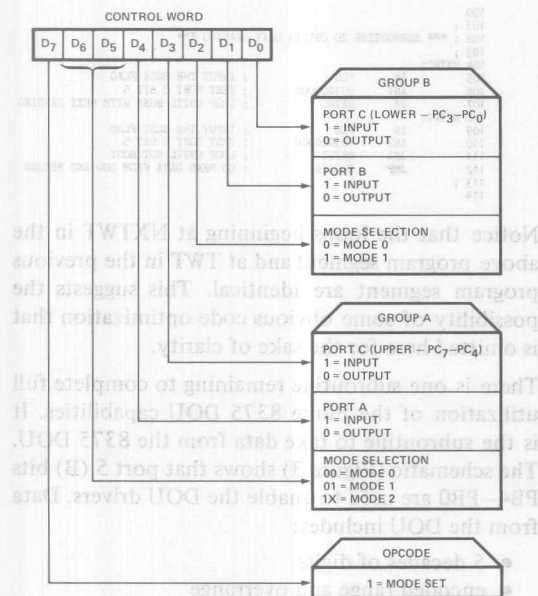
```

25 ;
26 ;
27 GETCH EQU 0220H ; GET CONSOLE INPUT, MASK OFF PARITY
28 CD EQU 01E8H ; CONSOLE OUTPUT
29 CROUT EQU 01F3H ; PRINT <CR><LF>
30 NMOUT EQU 02C2H ; DISPLAY BYTE IN ACCUM
31 ;
32

```

The use of the iSBC 80/10A parallel I/O ports requires that the mode of operation be defined for each port. This is typically done by an initialization subroutine executed when the iSBC 80/10A is powered up or reset.

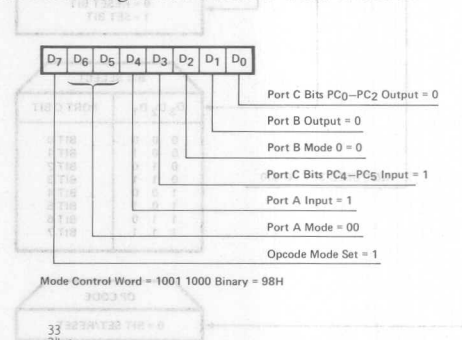
8255 Control Word. When the opcode field (bit 7) of a control word directed to an 8255 is equal to one, the control word is interpreted as a mode definition control word. The mode definition control word format is shown below:



Observing the schematic for the iSBC 80/10A — Fluke 8375 DOU (Figure 3), it can be seen that the 8255 #2 should be configured through the use of the mode control word as:

- Port 4 (A) Mode 0 Input
- Port 5 (B) Mode 0 Output
- Port 6 (C) Bits PC₂-PC₀ Output
- Port 6 (C) Bits PC₅-PC₄ Input

The following mode control word is used:



```

33 ;
34 ;
35 *** 8255 #2 INITIALIZATION SUBROUTINE
36 ;
37 INIT:
38 MVI A,10011000B ; LD MODE CONTROL WORD
39 OUT CWR ; OUTPUT TO 8255#2 CNTRL WD REG
40 ;
41

```

This coding loads the mode control word into the 8255 #2 control word register. Additional initialization code is required to set the strobe and trigger output ports to an inactive state. The schematic shows that inverting drivers have been used for both the strobes and the trigger. When a command is issued to place port 5 (B) into the output mode, bits PB₇-PB₀ are set to the low output state. Because the low outputs are then inverted and used as strobes to the Fluke 8375, they must then be disabled. The initialization subroutine concludes by disabling the strobes and trigger. The strobes are signals to the DOU which enable its drivers to send data to the iSBC 80/10A. The trigger is a signal to the DOU that the Fluke 8375 should take a reading.

```

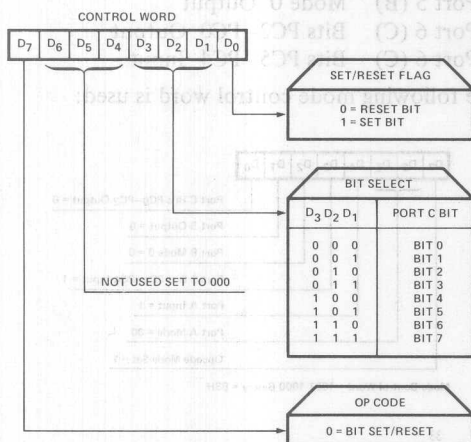
42 ;
43 ;
44 MVI A,0FFH ; LD MASK TO:
45 OUT STB ; DISABLE STROBES
46 OUT TRG ; DISABLE TRIGGER
47 RET
48 ;
49

```

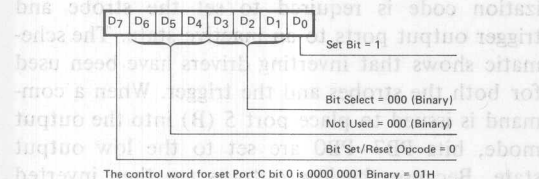
External Trigger Control. Two subroutines are implemented to enable and disable the external trigger mode of the instrument. These subroutines use the bit set/reset capability of the 8255 to independently set or reset three control lines of the Fluke 8375 DOU.

When the opcode field (bit 7) of an 8255 control word equals zero, the control word is a port 6 (C) bit set/reset command word.

The bit set/reset control word format is shown below:



The following example demonstrates how the port 6 (C) bit set/reset control word is constructed to disable the Fluke 8375 external trigger. Note from the schematic (Figure 3) that port 6 (C) bit 0 controls the inhibit external trigger line.



```

50 ;
51 ;
52 *** ENABLE EXTERNAL TRIGGER SUBROUTINE ***
53 ;
54 ETRIG:
55     MVI A,00000000B ; LD RESET BIT 0 CONTROL WORD
56     OUT CWR          ; OUTPUT TO 8255#2 CNTL WD REG
57     RET
58 ;
59 *** DISABLE EXTERNAL TRIGGER SUBROUTINE ***
60 ;
61 DTRIG:
62     MVI A,00000001B ; LD SET BIT 0 CONTROL WORD
63     OUT CWR          ; OUTPUT TO 8255#2 CNTL WD REG
64     RET
65 ;
66 ;

```

Subroutines to enable and disable the timeouts are written in an analogous fashion. The timeout enable line is controlled by port 6 (C) bit 2.

```

67 ;
68 ;
69 *** ENABLE TIMEOUTS SUBROUTINE ***
70 ;
71 EPOS:
72     MVI A,00000101B ; LD SET BIT 2 CONTROL WORD
73     OUT CWR          ; OUTPUT TO 8255#2 CNTL WD REG
74     RET
75 ;
76 *** DISABLE TIMEOUTS SUBROUTINE ***
77 ;
78 DPOS:

```

```

79 MVI A,00000100B ; LD RESET BIT 2 CONTROL WORD
80 OUT CWR          ; OUTPUT TO 8255#2 CNTL WD REG
81 RET
82 ;
83 ;

```

Obtaining Readings. The Fluke 8375 DOU allows readings to be taken in one of two modes. The first, a triggered mode, assumes that the external triggering has not been inhibited and requires the positive edge of a pulse with a minimum width of 1 microsecond on the trigger input. Setting and resetting the port 6 (C) bit 1 produces the 8375 external trigger. After a reading is triggered the 8375 busy flag is tested until the not busy state is reached. At that time the reading that was triggered can be read by the iSBC 80/10A. The last statement in this routine jumps to TKDATA which reads the data from the DOU and then executes the return.

```

84 ;
85 ;
86 *** SUBROUTINE TO TAKE EXTERNALLY TRIGGERED READING ***
87 ;
88 TRGR:
89     MVI A,00000010B ; LD RESET BIT 1 CONTROL WORD
90     OUT CWR          ; OUTPUT TO 8255#2 CNTL WD REG
91     INR A            ; MODIFY CONTROL WORD TO SET BIT 1
92     OUT CWR          ; OUTPUT TO 8255#2 CNTL WD REG
93 TWT:
94     IN FLG           ; INPUT THE BUSY FLAG
95     ANI 00100000B   ; TEST PORT C BIT 5
96     JNZ TWT          ; LOOP UNTIL NOT BUSY
97     JMP TKDATA       ; GO READ DATA FROM DOU AND RETURN
98 ;
99 ;

```

The second method for reading the Fluke 8375 is to rely on the sample rate set from the front panel controls and to wait until a full transition of the busy flag is observed. This guarantees that a previous reading is not mistakenly interpreted as a new reading.

```

100 ;
101 ;
102 *** SUBROUTINE TO OBTAIN NEXT READING ***
103 ;
104 NXTRD:
105     IN FLG           ; INPUT THE BUSY FLAG
106     ANI 00100000B   ; TEST PORT C BIT 5
107     JZ NXTRD         ; LOOP UNTIL BUSY WITH NEXT READING
108 NXTWT:
109     IN FLG           ; INPUT THE BUSY FLAG
110     ANI 00100000B   ; TEST PORT C BIT 5
111     JNZ NXTWT        ; LOOP UNTIL NOT BUSY
112     JMP TKDATA       ; GO READ DATA FROM DOU AND RETURN
113 ;
114 ;

```

Notice that the loops beginning at NXTWT in the above program segment and at TWT in the previous program segment are identical. This suggests the possibility of some obvious code optimization that is omitted here for the sake of clarity.

There is one subroutine remaining to complete full utilization of the Fluke 8375 DOU capabilities. It is the subroutine to take data from the 8375 DOU. The schematic (Figure 3) shows that port 5 (B) bits PB4-PB0 are used to enable the DOU drivers. Data from the DOU includes:

- 5 decades of digits
- encoded range and overrange

- function: Volts DC, Volts AC, Ohms, Kilohms
- modifiers: Filter, Ext. Ref., Remote
- overload
- trigger

The function of this subroutine is to read five bytes of data from the 8375 DOU and place them in a RAM buffer on the iSBC 80/10A.

```

115 ;
116 ;
117 ; *** SUBROUTINE TO TAKE DATA FROM 8375 DOU ***
118 ;
119 TKDATA: LXI H,RDBUF ; LD BUFFER POINTER
120 MVI A,0EFH ; SETUP FIRST STROBE
121 TK0: MOV B,A ; SAVE CURRENT STROBE
122 OUT STB ; STROBE DECADE PAIR
123 IN DATIN ; READ DATA
124 MOV M,A ; PLACE DATA INTO SBC 80/10 RAM
125 INX H ; INCREMENT BUFFER POINTER
126 MOV A,B ; RESTORE STROBE
127 RRC ; ROTATE TO NEXT STROBE POSITION
128 JC TK0 ; LOOP UNTIL BIT 0 STROBE DONE
129 OUT STB ; DISABLE ALL STROBES
130 RET
131 ;
132 ;
133 ;
134 ;

```

This completes the software required to service the Fluke 8375 DOU. The following code consists of a routine to display the data from the interface on the console output device and a short executive program to allow exercising of the driver subroutines.

The display subroutine takes 5 bytes of data from the RAM buffer in which the reading has been stored and prints them, 2 ASCII characters per 8-bit byte, on the console.

```

135 ;
136 ;
137 ; *** SUBROUTINE TO DISPLAY READING BUFFER ON CONSOLE ***
138 ;
139 DISPLAY: LXI H,RDBUF ; LD BUFFER POINTER
140 MVI D,5 ; INITIALIZE COUNTER
141 DISPO: MOV A,M ; LD NEXT BYTE FROM BUFFER
142 CALL NMOUT ; CALL SBC 80/10 MONITOR SUBROUTINE
143 ; TO DISPLAY ACCUMULATOR CONTENTS
144 INX H ; INCREMENT BUFFER POINTER
145 DCR D ; DECREMENT COUNTER
146 JNZ DISPO ; LOOP FOR 5 DISPLAY BYTES
147 RET
148 ;
149 ;
150 ;
151 ;

```

Operator Interface. The short executive program provides a tool for the purposes of exercising the 8375 DOU driver subroutines. The executive begins by calling the initialization subroutine and then continues on to prompt the operator with a '>' on the console. At that point the operator may enter one of the following characters, causing the program to execute the specified subroutine:

SUBR	DESCRIPTION
T ETRIG	Enable external trigger
I DTRIG	Disables external trigger
E EPOS	Enable programmed timeouts
D DPOS	Disable programmed timeouts
N NXTRD	Next reading
S TRGR	Trigger and get a reading
X DISPLAY	Display reading buffer

After the operator has entered a command character, the program obtains the address of the subroutine to be executed and proceeds to set up a return address on the stack. This technique allows a load program counter instruction (PCHL) to be used to enter the subroutine and a return instruction (RET) to resume execution of the executive.

```

152 ;
153 ;
154 ; *** SIMPLE EXECUTIVE EXERCISE PROGRAM ***
155 ;
156 START: LXI SP,STACK ; SETUP STACK POINTER
157 CALL INIT ; INITIALIZE THE SBC 80/10 8255#2
158 EXEC: CALL CROUT ; EXEC ENTRY POINT - PRINT <CR><LF>
159 MVI C,'>' ; C LOADED WITH PROMPT CHARACTER
160 CALL GETCH ; GET CMD CHAR, MASK OFF PARITY
161 CALL CO ; PRINT THE CHARACTER ON THE CONSOLE
162 MOV A,C ; PUT CHARACTER BACK INTO THE ACCUM
163 LXI B,NMDS ; C CONTAINS LOOP AND INDEX COUNT
164 H,CTAB ; HL POINTS TO CMDND TABLE
165 EXEC0: CMP M ; COMPARE TABLE ENTRY AND CHARACTER
166 JZ EXEC1 ; BRANCH IF EQUAL - CMDND RECOGNIZED
167 INX H ; ELSE, INCREMENT TABLE POINTER
168 DCR C ; DECREMENT LOOP COUNT
169 JNZ EXEC0 ; BRANCH IF NOT AT TABLE END
170 JMP EXEC ; ELSE, CMDND ILLGAL - IGNORE IT
171 EXEC1: LXI H,CADR ; LD ADR OF TABLE OF CMDND SUBRS
172 DAD B ; ADD WHAT IS LEFT OF LOOP COUNT
173 MOV A,M ; GET LSP OF ADR OF TABLE ENTRY TO A
174 INX H ; POINT TO NXT BYTE IN TABLE
175 MOV H,M ; GET MSP OF ADR OF TABLE ENTRY TO H
176 MOV L,A ; PUT LSP OF ADR OF TABLE ENTRY TO L
177 LXI D,EXEC ; SETUP RETURN ADR ON THE STACK
178 PUSH D
179 PCHL ; NEXT INSTR COMES FROM CMDND SUBR
180 ;
181 ;
182 ;
183 ;
184 ;
185 ;
186 ;
187 ;

```

The command and address tables as well as the reading buffer follow to complete the application software.

```

188 ;
189 ;
190 ; COMMAND AND ADDRESS TABLES
191 ;
192 CTAB: DB 'XSNDIT' ; NUMBER OF VALID COMMANDS
193 EQU $-CTAB
194 NMDS EQU $-CTAB
195 ;
196 CADR: DW 0
197 DW ETRIG
198 DW DTRIG
199 DW EPOS
200 DW DPOS
201 DW NXTRD
202 DW TRGR
203 DW DISPLAY
204 ;
205 ;
206 ; READING BUFFER AND STACK SPACE
207 ;
208 RDBUF: DS 5 ; READING BUFFER
209 ;
210 ;
211 ;
212 ;
213 END START ; TRANSFER ADDRESS IS TO START

```

SUMMARY/CONCLUSIONS

This instrumentation control application has been presented to demonstrate the simple techniques used to apply the iSBC 80/10A to the task of interfacing instrumentation. A natural extension of this example would include the control of the Fluke 8375 RCU, as well as the control of many additional instruments to build a small ATE system.

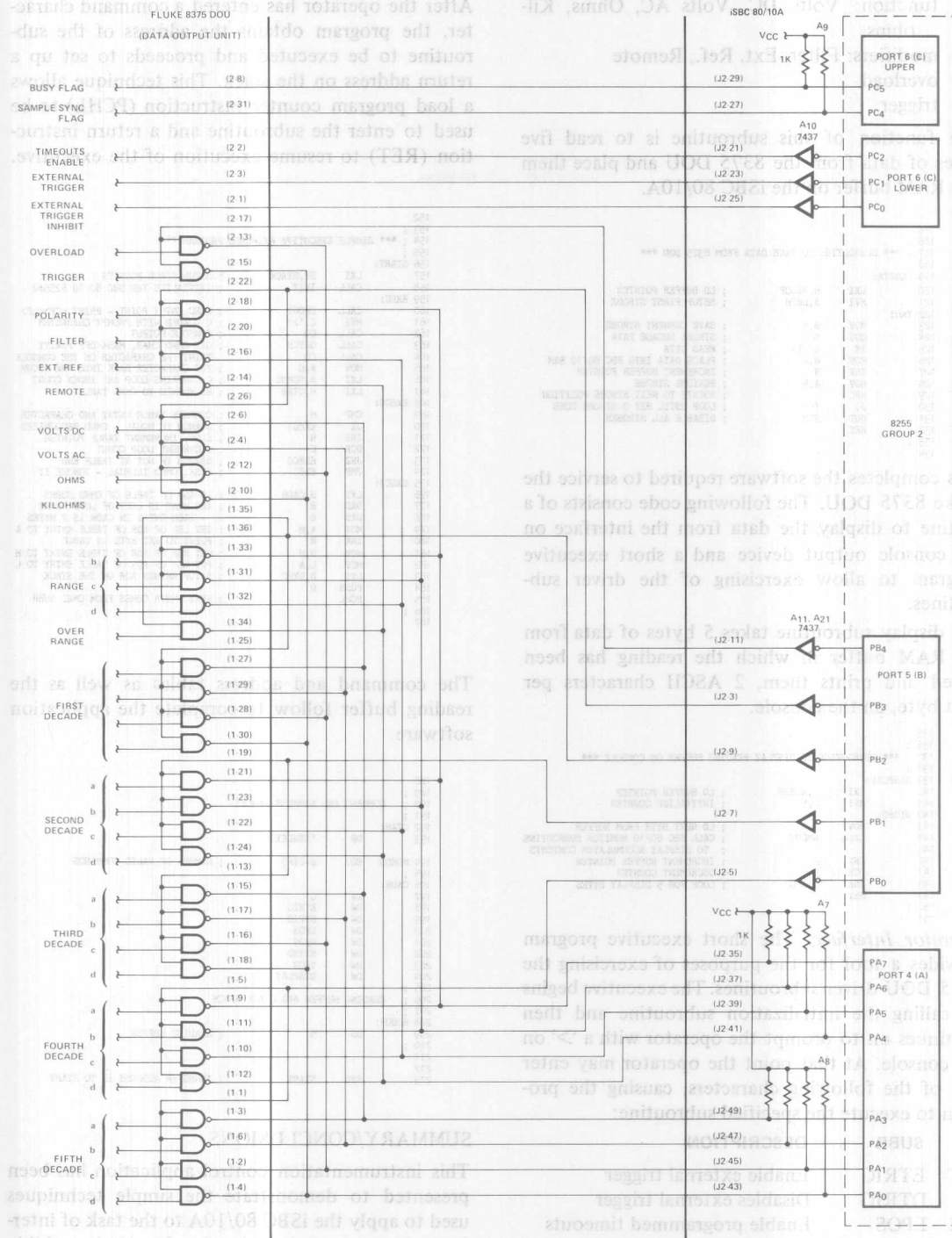


Figure 3: Interface Schematic

COMMUNICATION

A diverse range of single board computer applications exists in the field of communication. The increase in distributed processing generates requirements for self-contained computers to control elements of a communication system, increasing both the throughput and reliability.

There are many situations that necessitate monitoring and controlling a system from a remote site. Typical examples are systems that cover large geographic areas or systems in dangerous environments for human operators. If the object system, which provides the actual parallel inputs and outputs to the plant, is far from the controlling system, you can lower costs by reducing the number of interconnecting wires via the addition of multiplexers to both systems. In the extreme (and often desirable) case of reducing the interconnects to an absolute minimum, all communication between the systems takes place on a single serial data link. If large distances are involved, this link can be standard telephone wires. For moderate distances, the link can be a single twisted pair. In either case, the equipment used to interface the object system to the serial link is called a supervisory control and data acquisition (SCADA) terminal.

The decision to replace a multitude of interconnects with a SCADA terminal is largely economic. Cables and their associated drivers and receivers can represent a significant part of the total cost of a factory automation project, particularly if an electrically noisy environment requires the use of shielded cables. Any potential savings in cabling must, of course, compensate for the additional cost incurred by adding the SCADA terminal to the system.

Communication Application Example

A SCADA terminal demonstrates an industrial communication application of the iSBC 80/10A. The Intel® 8251 USART provides the serial communication link and the two Intel 8255 Programmable Parallel I/O devices provide 48 parallel lines for the object system. A block diagram of a SCADA terminal is shown in Figure 4.

The task of the software in this SCADA terminal example is two-fold. First, it must continually scan its parallel inputs, transmitting the status of those lines in a bit serial mode using the USART. And second, it receives bit serial data from the USART which is to be used to update the parallel outputs. Thus, a major portion of the software deals with

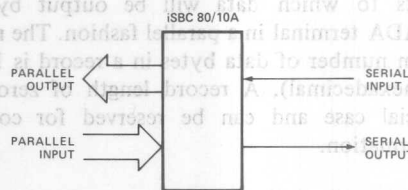


Figure 4. SCADA Terminal Block Diagram

the communications protocol on the serial data lines.

Communications Protocol. A communication protocol is an agreement between communications users that defines the record formats used for data transmissions. The protocol selected for this SCADA terminal application provides the following features:

1. A readable character set to simplify the human interface.
2. Error detection by means of a checksum.
3. Each record specifies the number of data bytes in the record and the initial port number.

Despite its value for human interface, the ASCII character set has problems representing 8-bit binary values, since the high-order bit is not used. Therefore, each binary value is treated as two 4-bit hexadecimal values. Because hexadecimal numbers fall in the range 0-9 and A-F, they can be represented as ASCII characters. However, this representation requires twice as many bytes as a pure binary format.

Record Format. The information encoded into the ASCII hexadecimal format is grouped to form records. Each record has a record mark to flag the beginning of the record, a number of ports specification (record length), destination output start port number, the data field itself, and a checksum.

The record format described below is according to the fields in the record.

Record mark field: Byte 0

The ASCII code for a colon (:) is used to signal the start of a record.

Number of ports field: Byte 1

The number of data bytes in the record is represented by a single ASCII hexadecimal digit in this field. This corresponds to the number of 8-bit

ports to which data will be output by the SCADA terminal in a parallel fashion. The maximum number of data bytes in a record is 15 (F in hexadecimal). A record length of zero is a special case and can be reserved for control information.

Port address field: Byte 2

The single ASCII hexadecimal digit in byte 2 gives the port number of the initial output port. The first data byte is output to the port indicated by the port address; successive bytes are output in successive port locations on the iSBC 80/10A or on expansion I/O boards.

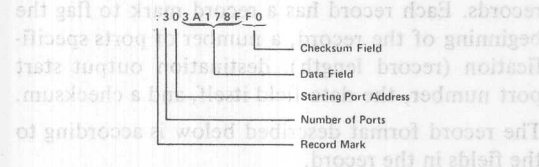
Data field: Bytes 3 to $3+2*(\text{number of ports})-1$

An 8-bit binary value is represented by two bytes containing the ASCII characters 0-9 or A-F, which represent a hexadecimal value between 0 and FF (0 and 255 decimal). The high-order digit is in the first byte of each pair.

Checksum field: Bytes $3+2*(\text{number of ports})$ to $3+2*(\text{number of ports})+1$

The checksum field contains the ASCII hexadecimal representation of the two's complement of the 8-bit sum of the 8-bit bytes that result from converting each pair of ASCII hexadecimal digits to one byte of binary, from the number of ports field (the number of ports and port address constitute a pair) to and including the last byte of the data field. Therefore, the sum of all the ASCII pairs in a record after converting to binary, from the number of ports field to and including the checksum field, is zero.

Sample Hexadecimal format:



Design Approach Using a State Diagram. Before proceeding to examine the software used to implement the SCADA terminal, consider how the problem might have been approached with TTL logic rather than a microcomputer. The design would likely have been formulated with a state diagram to specify the transitions of a sequential state machine. The sequential-circuit operations would include decoding the serial input records and

encoding the serial output records. An examination of the serial input record state diagram (Figure 5) is useful in understanding some of the procedures encountered later.

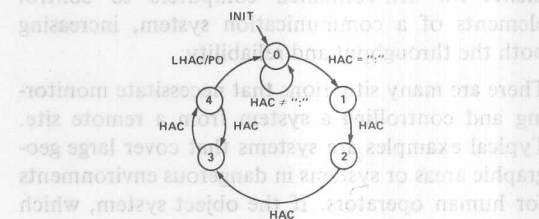


Figure 5. State Diagram

Notes: HAC = Hexadecimal ASCII character
LHAC = Last Hexadecimal ASCII character
PO = Parallel output

The receipt of an invalid HAC will cause a return to state 0.

The receipt of a colon at any time will produce a transition to state 1.

STATE	DESCRIPTION
0	= record mark state
1	= number of ports state
2	= start port number state
3	= high-order half of data byte state
4	= low-order half of data byte state

State 0 is entered at the time of initialization. All state transitions occur when the next character is received. States 1, 2, and 3 are entered with the input of a colon (:), the number of ports and start port number, respectively. States 3 and 4 will cycle as required until all the high and low-order pairs of data have been input. The transition from state 4 to state 0 occurs when the last data byte has been received. If the checksum is correct, the parallel output latches are loaded with the data field of the record.

There are many references to the states contained in this diagram during the discussion of the software procedures. Thus, the state diagram is used as a "flowchart" for the software. As in the other examples in this application note, a textual description accompanies each segment of code. Intel's high-level programming language, PL/M-80, has been used to show the capability to program in a natural, algorithmic language which eliminates the need to manage register usage or memory allocation.

SCADA Terminal Program. The program begins with a comment, that is followed by the program segment label "SCADA". With resident PL/M-80, all programs are considered to be labelled blocks, or modules. This means that all PL/M programs must begin with a LABEL and a DO statement and end with an END statement.

```

/*
  INDUSTRIAL COMMUNICATION APPLICATION

  SCADA TERMINAL

1  SCADA:
  DO;

```

All variables used in the program must be declared before they can be referred to by their identifiers. This is done by means of a DECLARE statement. In addition to the declaration of variables, macros are declared using the reserved word LITERALLY. These macros are expanded at compile time by textual substitution.

```

2 1 DECLARE
  SRL$IN$STATE BYTE,
  SRL$IN$PRT BYTE,
  SRL$IN$CNT BYTE,
  PRL$IN$STATE BYTE,
  PRL$IN$STRT$PRT BYTE,
  PRL$IN$NMBS$PRTS BYTE,
  SRL$IN$PRL$OUT$BFR(3) BYTE,

  PRL$OUT$PRT$0 LITERALLY '0ESH',
  PRL$OUT$PRT$1 LITERALLY '0EAH',
  PRL$OUT$PRT$2 LITERALLY '0EBH',

  SRL$OUT$STATE BYTE,
  SRL$OUT$PRT BYTE,
  SRL$OUT$CNT BYTE,
  PRL$OUT$STATE BYTE,
  PRL$OUT$STRT$PRT BYTE,
  PRL$OUT$NMBS$PRTS BYTE,
  SRL$OUT$PRL$IN$BFR(4) BYTE,

  PRL$IN$PRT$0 LITERALLY '0E5H',
  PRL$IN$PRT$1 LITERALLY '0E6H',
  PRL$IN$PRT$2 LITERALLY '0E7H',

  USART$CMD LITERALLY '0EDH',
  USART$IN LITERALLY '0ECH',
  USART$OUT LITERALLY '0ECH',
  USART$STATUS LITERALLY '0EDH',
  USART$MODE$INSTR LITERALLY '0CEH',
  USART$CMD$INSTR LITERALLY '025H',

  TXRDY LITERALLY '001H',
  RXRDY LITERALLY '002H',

  PPI$CWR$1 LITERALLY '0E7H',
  PPI$CWR$2 LITERALLY '0EBH',
  PPI$CWD$1 LITERALLY '080H',
  PPI$CWD$2 LITERALLY '09BH',

  TRUE LITERALLY '0FFH',
  FALSE LITERALLY '000H',

  FOREVER LITERALLY 'WHILE TRUE',

  NEXT$BYTE BYTE,
  CHECKSUM BYTE;

```

8251 and 8255 Initialization. The INIT procedure sets up the 8251 and 8255's and initializes several variables. Interrupts are disabled to insure that no interrupts are serviced during the execution of the INIT procedure.

```

3 1 INIT: PROCEDURE;
4 2   DISABLE;

```

The serial input and serial output state counters are set to state 0. Port number 0 is the parallel input start port and 3 ports of data are input from the parallel ports for serial transmission.

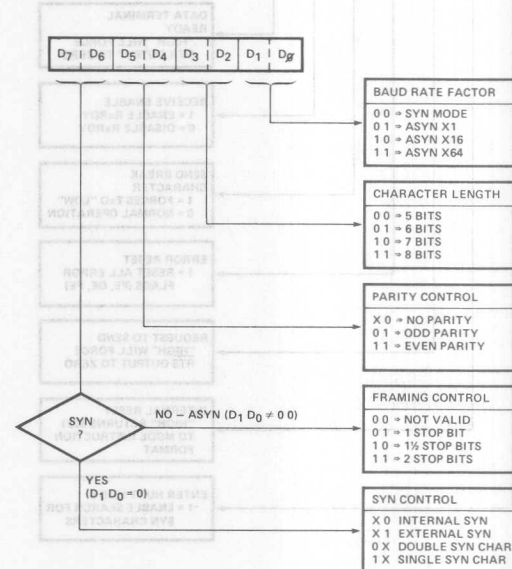
```

5 2   SRL$IN$STATE = 0;
6 2   SRL$OUT$STATE = 0;
7 2   PRL$IN$STRT$PRT = 0;
8 2   PRL$IN$NMBS$PRTS = 3;

```

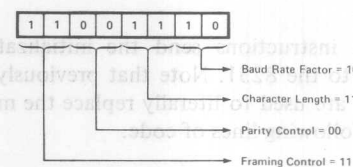
The Intel 8251 USART must be set up by loading it with mode and command instructions.

The mode instruction format is shown below:



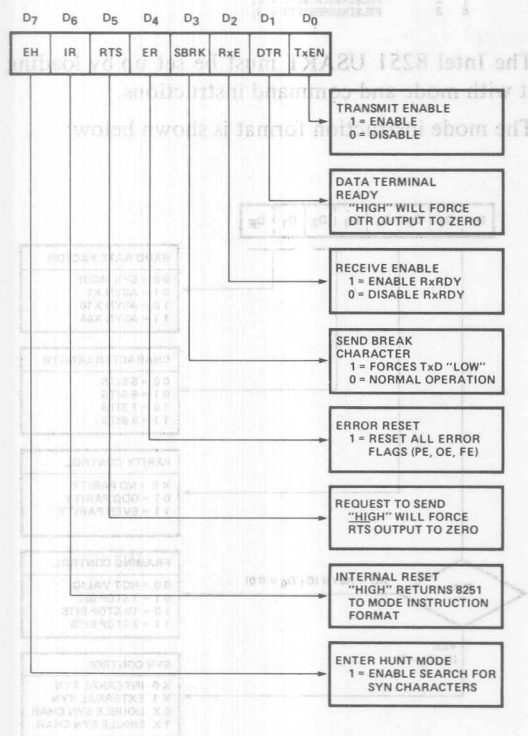
The 8251 characteristics required by this SCADA terminal application include 9600 baud transmission and 8-bit characters. The parallel inputs of the 8255's are periodically scanned. The scanning frequency is determined by the baud rate and number of ports of data being transmitted. For example, the transmission of 3 ports of data requires 11 characters. At a baud rate of 9600 the approximate scan rate is 100 Hz.

The following 8251 mode instruction is used:



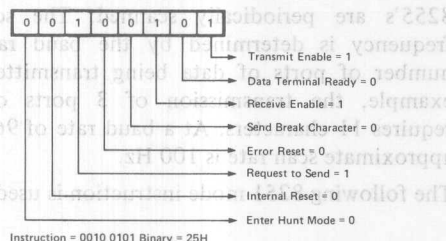
After the mode instruction is sent to the 8251, a command instruction is required to complete the 8251 initialization.

The command instruction format is shown below:



The command instruction enables the transmit and receive functions of the 8251.

The following command instruction is used:



Output instructions send the initialization commands to the 8251. Note that previously declared macros are used to literally replace the mnemonics in the following lines of code.

```
9 2 OUTPUT(USART$CMD) = USART$MODE$INSTR;
10 2 OUTPUT(USART$CMD) = USART$CMD$INSTR;
```

Initialization of the 8255's is then done to set up the following configurations:

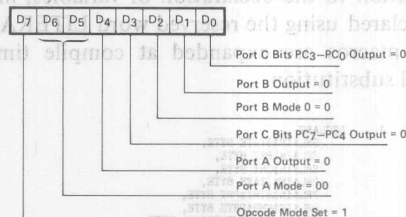
8255 #1

Port 1 (A) Mode 0 Output
Port 2 (B) Mode 0 Output
Port 3 (C) Mode 0 Output

8255 #2

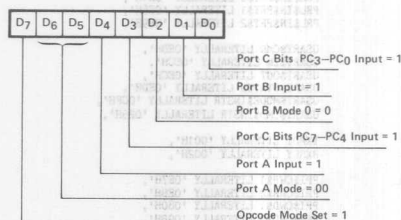
Port 4 (A) Mode 0 Input
Port 5 (B) Mode 0 Input
Port 6 (C) Mode 0 Input

The following command instruction is used for the 8255 #1:



Mode Control Word = 1000 0000 Binary = 80H

The following command instruction is used for the 8255 #2:



Mode Control Word = 1001 1011 Binary = 9BH

The 8255 initialization commands are given in a similar manner to the 8251 commands.

The INIT procedure concludes by enabling interrupts.

```
13 2 ENABLE;
14 2 END INIT;
```

Conversion Procedures. Two conversion procedures are required in the program. The first procedure produces a hexadecimal ASCII character from a 4-bit binary value. A typed procedure has been used which returns a value of the type byte. It is called by using its name in an expression.

```

15 1  CHAR$CONV: PROCEDURE (CHAR) BYTE;
16 2  DECLARE CHAR BYTE;
17 2  CHAR = CHAR * '0';
18 2  IF CHAR > '9' THEN
19 2  CHAR = CHAR + 7;
20 2  RETURN CHAR;
21 2  END CHAR$CONV;

```

The second procedure produces a 4-bit binary value from a hexadecimal ASCII character. Because this procedure is used only when a hexadecimal ASCII character is expected, an illegal character (i.e., not a 0-9 or A-F) causes the serial input state counter to indicate state 0. This procedure is also typed. The NMB\$CONV procedure emphatically illustrates the point that PL/M-80 performs unsigned arithmetic. Note that when the ASCII value for a zero is subtracted from the digit, NUM = NUM - '0'; a positive number is always produced, even if the value of NUM is less than '0'.

```

22 1  NMB$CONV: PROCEDURE (NMB) BYTE;
23 2  DECLARE NMB BYTE;
24 2  NMB = NMB - '0';
25 2  IF NMB > 9 THEN
26 2  DO;
27 3  IF (NMB > 16) AND (NMB < 23) THEN
28 3  NMB = NMB - 7;
29 3  SRL$IN$STATE = 0;
30 3  END;
31 2  RETURN NMB;
32 2  END NMB$CONV;

```

Parallel Input Procedure. A parallel input procedure is used to input data bytes from the 8255's. The data bytes are then transmitted by the bit serial output device. This procedure also computes the checksum for the serial output record. The checksum, TEMP2, is initialized to contain the parallel input number of ports and the start port, shifted to fit within a single byte. Each cycle of the iterative DO block adds the next data byte to the checksum and places the input data into the SRL\$OUT\$PRL\$IN\$BFR array until the loop is complete. The checksum is then computed as the two's complement of the accumulated sum and also stored in the serial input parallel output buffer.

```

33 1  PARALLEL$IN: PROCEDURE;
34 2  DECLARE (TEMP1,TEMP2) BYTE;
35 2  TEMP2 = PRL$IN$NMB$PRTS * 16 + PRL$IN$STRT$PRT;
36 2  DO PRL$IN$STATE = PRL$IN$STRT$PRT TO
    PRL$IN$STRT$PRT + PRL$IN$NMB$PRTS - 1;
37 3  DO CASE PRL$IN$STATE;
38 4  /* PRL IN PRT 0 */
    TEMP1 = INPUT(PRL$IN$PRT0);
39 4  /* PRL IN PRT 1 */
    TEMP1 = INPUT(PRL$IN$PRT1);
40 4  /* PRL IN PRT 2 */
    TEMP1 = INPUT(PRL$IN$PRT2);
41 4  END;
42 3  SRL$OUT$PRL$IN$BFR(PRL$IN$STATE) = TEMP1;
43 3  TEMP2 = TEMP2 + TEMP1;
44 3  END;
45 2  SRL$OUT$PRL$IN$BFR(PRL$IN$STRT$PRT + PRL$IN$NMB$PRTS) = -TEMP2;
46 2  END PARALLEL$IN;

```

Parallel Output Procedure. When a complete serial input record has been received and the checksum is correct, the transition from state 4 to state 0 is accompanied by the parallel output of the data from the data field of the serial input record. The parallel output starting port and the number of ports of data is contained in the input record and is thus used in directing the parallel output operation. An iterative DO block increments the PRL\$OUT\$STATE index variable through the required ports and a DO CASE block selectively executes one of the OUTPUT statements for each cycle of the loop.

```

47 1  PARALLEL$OUT: PROCEDURE;
48 2  DECLARE TEMP BYTE;
49 2  DO PRL$OUT$STATE = PRL$OUT$STRT$PRT TO
    PRL$OUT$STRT$PRT + PRL$OUT$NMB$PRTS - 1;
50 3  TEMP = SRL$IN$PRL$OUT$BFR(PRL$OUT$STATE);
51 3  DO CASE PRL$OUT$STATE;
52 4  /* PRL OUT PRT 0 */
    OUTPUT(PRL$OUT$PRT0) = TEMP;
53 4  /* PRL OUT PRT 1 */
    OUTPUT(PRL$OUT$PRT1) = TEMP;
54 4  /* PRL OUT PRT 2 */
    OUTPUT(PRL$OUT$PRT2) = TEMP;
55 4  END;
56 3  END;
57 2  END PARALLEL$OUT;

```

Serial Input and Output Procedures. The next two procedures contain the software implementations of the state diagram described previously. The processing during each state of the first procedure, the serial character input procedure, is described in the following text.

The procedure begins by reading a character from the 8251 and then converts the character into a 4-bit binary value using the number conversion procedure. The DO CASE block is the mechanism by which a program segment is selected to examine

the input character, provide the required outputs, and to specify the transition to the next state.

```

58 1 SERIAL$CHAR$IN; PROCEDURE;
59 2 DECLARE (CHAR,TEMP) BYTE;
60 2 CHAR = INPUT(USART$IN) AND 07FH;
61 2 TEMP = NMB$CONV(CHAR);
62 2 DO CASE SRL$IN$STATE;

```

State 0 is entered through the initialization process, at the completion of the processing of a serial input record, or when an invalid character has been received. The serial input state will remain 0 until a colon (:) is received, at which time a transition to state 1 is specified.

```

63 3 /* SRL IN STATE 0 = RECORD MARK */
64 4 DO;
65 4 IF CHAR = ':' THEN
66 4 SRL$IN$STATE = 1;

```

The parallel output number of ports is obtained, the counter initialized, and a transition to state 2 is specified from state 1.

```

67 3 /* SRL IN STATE 1 = NMB PRTS */
68 4 DO;
69 4 PRL$OUT$NMB$PRTS = TEMP;
70 4 SRL$IN$CNT = TEMP;
71 4 SRL$IN$STATE = 2;

```

In state 2, the parallel output starting port number is obtained, the serial input port is initialized, the checksum is set to contain the parallel output number of ports and starting port, and a transition to state 3 is specified.

```

72 3 /* SRL IN STATE 2 = STRT PRT */
73 4 DO;
74 4 PRL$OUT$STRT$PRT = TEMP;
75 4 SRL$IN$PRT = TEMP;
76 4 CHECKSUM = PRL$OUT$NMB$PRTS*16 + PRL$OUT$STRT$PRT;
77 4 SRL$IN$STATE = 3;

```

In state 3 the high-order half of a data byte is obtained and shifted into the proper position of the NEXT\$BYTE variable. A transition is specified to state 4.

```

78 3 /* SRL IN STATE 3 = HI ORDER HALF DATA BYTE */
79 4 DO;
80 4 NEXT$BYTE = TEMP*16;
81 4 SRL$IN$STATE = 4;

```

State 4 is the final state and requires more processing than the others. First, a whole byte of data is assembled by adding the low and high-order data halves, and then testing to determine if the checksum has been received. If so, and the checksum is correct, the parallel output procedure is executed. Once the entire serial input record has been received, a transition is specified to state 0 whether the checksum is correct or not. However, if the

serial input count has not been exhausted, the assembled byte is placed into the serial input parallel output buffer and a transition back to state 3 is specified.

```

82 3 /* SRL IN STATE 4 = LO ORDER HALF DATA BYTE */
83 4 DO;
84 4 NEXT$BYTE = NEXT$BYTE + TEMP;
85 4 CHECKSUM = CHECKSUM + NEXT$BYTE;
86 4 IF SRL$IN$CNT = 0 THEN
87 4 DO;
88 4 IF CHECKSUM = 0 THEN
89 4 CALL PARALLEL$OUT;
90 4 SRL$IN$STATE = 0;
91 4 ELSE
92 4 DO;
93 4 SRL$IN$PRL$OUT$BFR(SRL$IN$PRT) = NEXT$BYTE;
94 4 SRL$IN$PRT = SRL$IN$PRT + 1;
95 4 SRL$IN$CNT = SRL$IN$CNT - 1;
96 4 SRL$IN$STATE = 3;
97 4 END;

```

```

98 3 END; /* END OF CASES */
99 2 END SERIAL$CHAR$IN;

```

The serial character output procedure is similar to the serial character input procedure. During state 0 the parallel inputs of the 8255's are stored in the serial output parallel input buffer for transmission.

```

100 1 SERIAL$CHAR$OUT; PROCEDURE;
101 2 DECLARE (CHAR,TEMP) BYTE;
102 2 CHAR = 0;
103 2 DO CASE SRL$OUT$STATE;

```

```

104 3 /* SRL OUT STATE 0 = RECORD MARK */
105 4 DO;
106 4 CHAR = ':';
107 4 CALL PARALLEL$IN;
108 4 SRL$OUT$STATE = 1;

```

```

109 3 /* SRL OUT STATE 1 = NMB PRTS */
110 4 DO;
111 4 TEMP = PRL$IN$NMB$PRTS;
112 4 SRL$OUT$CNT = TEMP;
113 4 SRL$OUT$STATE = 2;

```

```

114 3 /* SRL OUT STATE 2 = STRT PRT */
115 4 DO;
116 4 TEMP = PRL$IN$STRT$PRT;
117 4 SRL$OUT$PRT = TEMP;
118 4 SRL$OUT$STATE = 3;

```

```

119 3 /* SRL OUT STATE 3 = HI ORDER HALF DATA BYTE */
120 4 DO;
121 4 TEMP = SHR(SRL$OUT$PRL$IN$BFR(SRL$OUT$PRT),4);
122 4 SRL$OUT$STATE = 4;

```

```

123 3 /* SRL OUT STATE 4 = LO ORDER HALF DATA BYTE */
124 4 DO;
125 4 TEMP = SRL$OUT$PRL$IN$BFR(SRL$OUT$PRT) AND 0FH;
126 4 IF SRL$OUT$CNT = 0 THEN
127 4 SRL$OUT$STATE = 0;

```

```

128 4 ELSE
129 4 DO;
130 4 SRL$OUT$CNT = SRL$OUT$CNT - 1;
131 4 SRL$OUT$PRT = SRL$OUT$PRT + 1;
132 4 SRL$OUT$STATE = 3;
133 4 END;

```

```

134 3 END; /* END OF CASES */
135 2 IF CHAR <> ':' THEN
136 2 CHAR = CHAR$CONV(TEMP);
137 2 OUTPUT(USART$OUT) = CHAR;

```

```

138 2 END SERIAL$CHAR$OUT;

```

Interrupt Service Routine. The software in this SCADA terminal application example is interrupt driven. Interrupts, which occur when the transmitter of the 8251 is ready for another character or when the receiver has obtained a serial character, direct the execution of either the serial input

or output character procedures. The following procedure is entered when an interrupt occurs.

```

138 1  USART$INTERRUPT: PROCEDURE INTERRUPT 7;
139 2  DECLARE STATUS BYTE;
140 2  STATUS = INPUT(USART$STATUS);
141 2  IF (STATUS AND TXRDY) = TXRDY THEN
142 2  CALL SERIAL$CHAR$OUT;
143 2  IF (STATUS AND RXRDY) = RXRDY THEN
144 2  CALL SERIAL$CHAR$IN;
145 2  END USART$INTERRUPT;

```

Main Program. The function of the main program is rather simple. It calls the initialization routine and then loops "FOREVER." Notice that the other software is executed only when an interrupt occurs. Rather than loop idly while waiting for an interrupt, the "main program" could take advantage of excess CPU time by processing some other task.

```

MAIN$PROGRAM:
*****
146 1  CALL INIT;
147 1  DO FOREVER;
148 2  END;

```

```

149 1  END;

```

SUMMARY/CONCLUSIONS

Further consideration should be given to error checking in the implementation of a SCADA terminal. A checksum has been used in this example which provides some error detection but no correction.

The industrial communication example in this application note has shown a SCADA terminal. Besides providing a convenient forum in which to explore the use of PL/M in an interrupt-driven environment, this application provides a realistic and almost fully-developed tool for the replacement of a multitude of parallel lines. Two such systems can be connected through the serial lines to provide a parallel to parallel transmission scheme as shown in Figure 6.

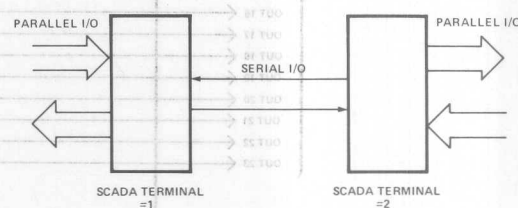


Figure 6. Two SCADA Terminals

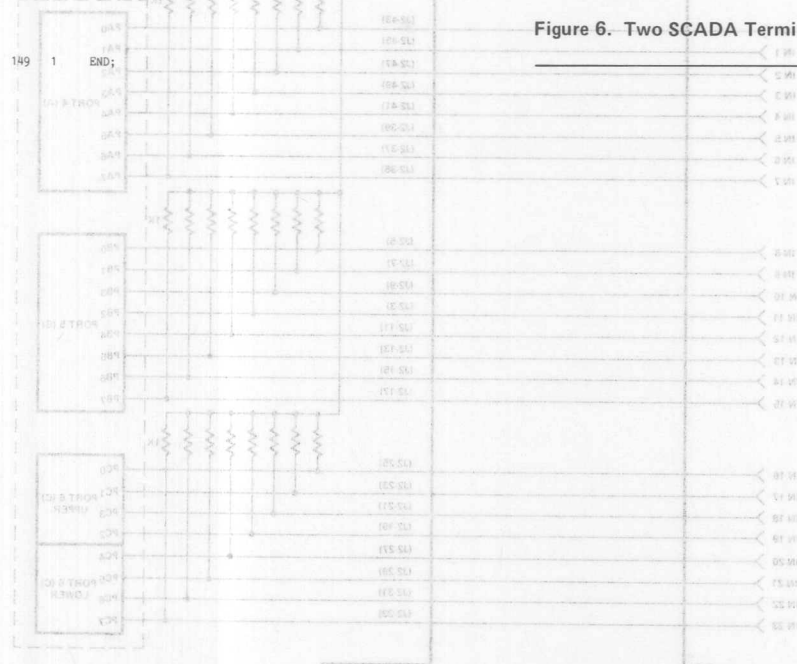


Figure 7. SCADA Terminal Schematic

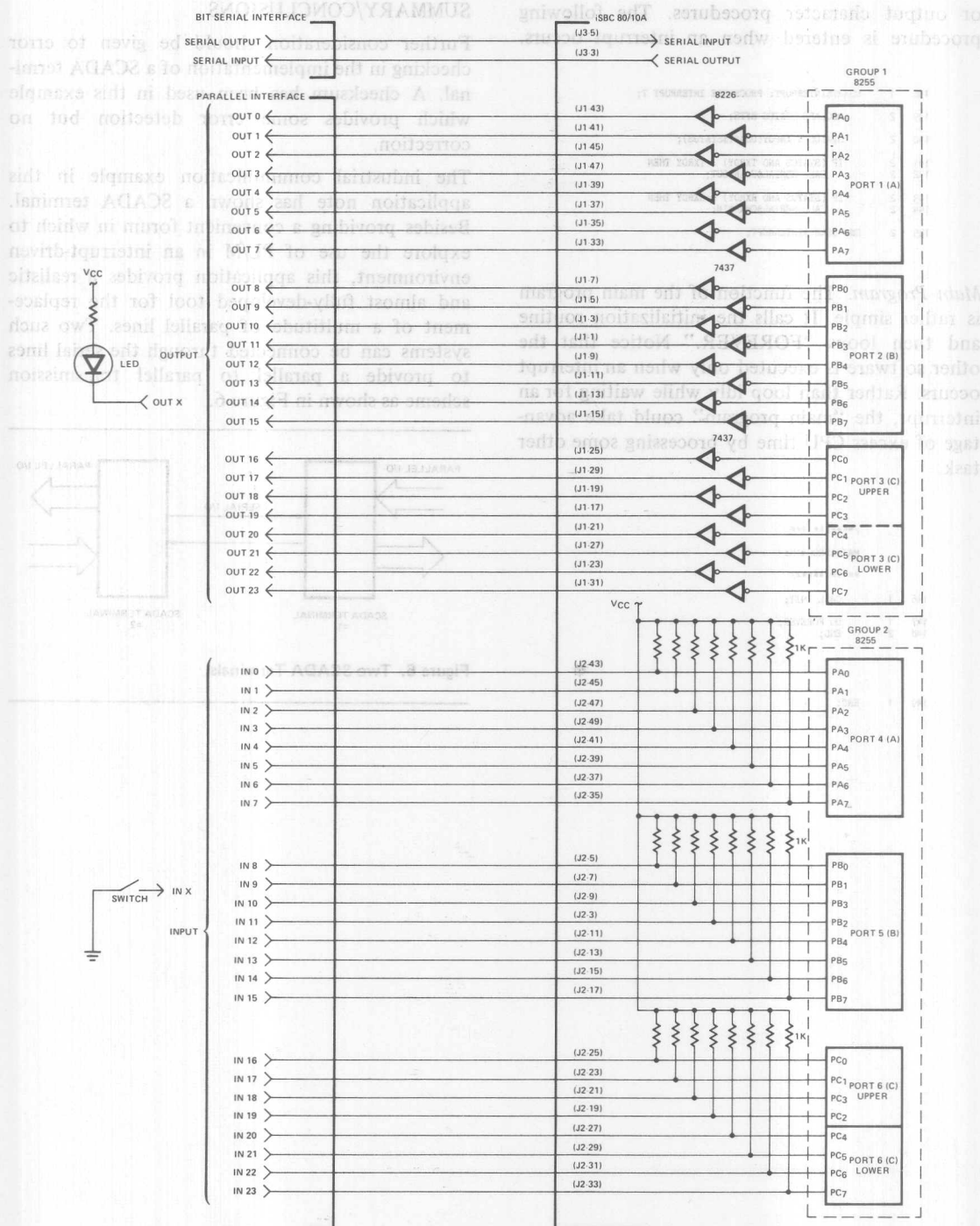


Figure 7. SCADA Terminal Schematic

PROCESS CONTROL

Many single board computers have already been applied in the field of process control. Some of the common denominators observed in these applications include the use of A/D and D/A peripheral boards, process monitoring functions such as servicing display panels for operator interaction, and alarm indicators.

Temperature Monitoring Application Example

A temperature monitoring system has been developed for the purposes of a process control application example. The single open loop system utilizes an A/D converter, a multiplexed display, switches for operator control, and two alarms. A block diagram of the operator's panel is shown in Figure 8 and a schematic in Figure 9.

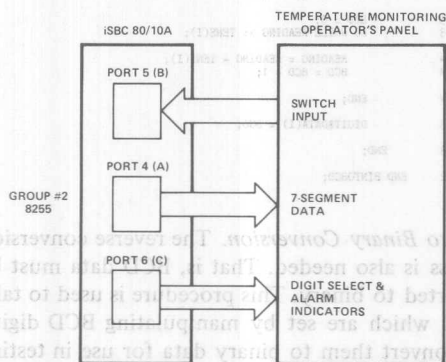


Figure 8. Operator's Panel Block Diagram

Operator's Panel. The operator's panel in this temperature monitoring system contains four 7-segment displays to show the temperature, two light emitting diodes (LEDs) that indicate alarm-low and alarm-high conditions, and six switches. The function of the switches is as follows:

Set Limit — controls whether the current temperature reading is to be displayed (off) or if upper/lower limits are to be set (on).

Set Hi Lo — when set limit is “on”, this switch controls whether the low (off) or high (on) limit is to be displayed.

Digit Selects — these two switches control the selection of the digit of the limit which is to be modified. The four binary positions 00, 01, 10 and 11 correspond to the four 7-segment digits.

Leave It — controls whether the digit selected is to be incremented (off) or maintained at its current value (on). When this switch is “off” the digit selected is incremented every 512 ms until the operator turns the switch “on”.

Enable Alarm – when set limit is “off” and the current temperature is displayed, this switch controls whether the action of the alarm indicators is to be enabled (on) or disabled (off).

The simple means used to set upper and lower temperature limits is similar to setting the time on a digital wrist watch.

The purpose of the software is to initialize the system and then to enter an endless loop which accumulates 16 readings, updates the displayed reading or handles limit setting, updates the display latches, waits 4 ms, and obtains an A/D reading.

Temperature Monitoring Program. This application example has been coded in Intel's resident PL/M-80 language.

[illegible]

The declaration statement includes some dimensioned variables with INITIAL attributes. They provide data strobe positions, a table of bit patterns to convert BCD data to 7-segment data, and a table of the powers of 10 for binary to BCD conversions.

```

2      1      DECLARE
      READING ADDRESS
      DIGITS(4) BYTE INITIAL (80H,40H,20H,10H),
      BC0T07SEG(11) BYTE INITIAL (3FH,06H,5BH,4FH,66H,
                                60H,7CH,07H,7FH,67H,0),
      TENS(4) ADDRESS INITIAL (1000,100,10,1),
      DIGITDATA(4) 3 BYTES,
      NXTDIGIT BYTE,
      UPDATE$COUNT BYTE,
      SET$COUNT BYTE,
      LIMIT(2) ADDRESS,
      ACCUM$COUNT ADDRESS,

      CWR LITERALLY 'OE8H',
      SLCIT LITERALLY 'OEAH',
      SSGS LITERALLY 'OE8H',
      SMTS LITERALLY 'OE8H',
      SETUP$PORTS LITERALLY 'OE2H',

      SET$LIMIT LITERALLY '001H',
      SET$ISLO LITERALLY '002H',
      LEAVE$IT LITERALLY '010H',
      DIGIT$SLCT LITERALLY '000H',
      ENBLE$ALARM LITERALLY '02H',
      SET$ALARM$LO LITERALLY '001H',
      SET$ALARM$HI LITERALLY '003H',
      RESET$ALARM$LO LITERALLY '000H',
      RESET$ALARM$HI LITERALLY '002H',

      TRUE LITERALLY '0FFH',
      FOREVER LITERALLY 'WHILE TRUE':

```


The analog to digital conversion procedure has been coded in assembly language and is not included in this application note. It is declared as an external typed procedure with no arguments and returns a value of the type address. The value returned is the current temperature. The ATOD procedure is linked later in a step to produce an absolute load module of the entire program.

```
3 1 ATOD: PROCEDURE ADDRESS EXTERNAL;
4 2 END ATOD;
```

Bit set/reset functions of the 8255 have been used to control the alarm-low and high output bits. Use of this function allows individual bits to be controlled without affecting others of port C which are concurrently selecting the digit to be multiplexed on the display.

```
5 1 RESET$ALARMS: PROCEDURE;
6 2 OUTPUT(CWR) = RESET$ALARM$LO;
7 2 OUTPUT(CWR) = RESET$ALARM$HI;
8 2 END RESET$ALARMS;
```

The following procedure is used to initialize the 8255 and several program variables.

```
9 1 INIT: PROCEDURE;
10 2 OUTPUT(CWR) = SETUP$PORTS;
11 2 CALL RESET$ALARMS;
12 2 NXT$DIGIT = 0;
13 2 UPDATE$COUNT = 0;
14 2 SET$COUNT = 7;
15 2 READING = 0;
16 2 ACCUM$RDNG = 0;
17 2 LIMIT(0) = 0000;
18 2 LIMIT(1) = 9999;
```

```
19 2 END INIT;
```

A multiplexed display is controlled by the software. Two ports of an 8255 are required for this function shown in Figure 9. The first output port holds the data which drives the four 7-segment displays in parallel. The second output port contains four strobes, each going to a separate common cathode of one of the 7-segment displays.

The update display procedure begins by blanking 7-segment data in the output port. This step avoids shadows that would be produced if the data for the next digit position were loaded prior to updating the strobe. The strobe is then advanced, retaining the alarm bits that occupy other bits of the same output port. Note that an output configured 8255 port can be read with an 8080A INPUT instruction to determine the currently latched output data. The BCD data is obtained from the next digit position of the DIGIT\$DATA array and used as a subscript into a table of BCDDTO7SEG data. The 7-segment data is also

output to the 8255 port in the same statement. The procedure concludes by advancing the NXT\$DIGIT pointer.

```
20 1 DISPLAY$UPDATE: PROCEDURE;
21 2 OUTPUT(SEGS) = 0;
22 2 OUTPUT(SLCT) = (DIGITS(NXT$DIGIT) OR (INPUT(SLCT) AND 03H));
23 2 OUTPUT(SEGS) = BCDDTO7SEG(DIGIT$DATA(NXT$DIGIT));
24 2 NXT$DIGIT = (NXT$DIGIT+1) AND 03H;
25 2 END DISPLAY$UPDATE;
```

Binary to BCD Conversion. Binary data from the A/D converter must be converted to BCD before it can be used by the DISPLAY\$UPDATE procedure to show the current temperature reading. The BINTOBCD procedure performs this conversion operation.

```
26 1 BINTOBCD: PROCEDURE;
27 2 DECLARE (BCD,I) BYTE;
28 2 DO I = 0 TO 3;
29 3 BCD = 0;
30 3 DO WHILE HEADING >= TENS(I);
31 4 READING = READING - TENS(I);
32 4 BCD = BCD + 1;
33 4 END;
34 3 DIGIT$DATA(I) = BCD;
35 3 END;
36 2 END BINTOBCD;
```

BCD to Binary Conversion. The reverse conversion process is also needed. That is, BCD data must be converted to binary. This procedure is used to take limits, which are set by manipulating BCD digits, and convert them to binary data for use in testing against current temperature readings. Based variables have been used in this procedure to allow access to the actual variables used as arguments in the calling program.

```
37 1 BCDBTOBIN: PROCEDURE (BCD$ARRAY$ADR,BIN$DATA$ADR);
38 2 DECLARE
  (BCD$ARRAY$ADR,BIN$DATA$ADR) ADDRESS,
  (BCD$ARRAY BASED BCD$ARRAY$ADR) (4) BYTE,
  (BIN$DATA BASED BIN$DATA$ADR) ADDRESS,
  I BYTE;
39 2 BIN$DATA = 0;
40 2 DO I = 0 TO 3;
41 3 BIN$DATA = BIN$DATA*10 + BCD$ARRAY(I) #/
42 3 BIN$DATA = SHL(BIN$DATA,1) + SHL(BIN$DATA,3) + BCD$ARRAY(I);
43 2 END;
43 2 END BCDBTOBIN;
```

Updating the Display. The UPDATE procedure is entered each time 16 readings have been taken from the A/D converter. The UPDATE\$COUNT is reset and the operator switches are input to control the execution path through the procedure. The accumulated reading, which is the total of 16 A/D readings, is divided by 16 to obtain an average reading. Then the accumulated reading is zeroed.

```

44 1  UPDATE: PROCEDURE;
45 2  DECLARE (SWT$FLG,HI$LO,DIGIT) BYTE;
46 2  UPDATES$COUNT = 15;
47 2  SWT$FLG = INPUT(SWT$);
48 2  READING = SHR(ACCUM$RDNG,4);
49 2  ACCUM$RDNG = 0;

```

Setting Limits. If the set limit switch is ON, the limits are to be dealt with instead of testing and displaying the current temperature reading. The alarms are reset during limit setting. The specified limit is converted to BCD and then the Leave-It switch is tested to see if the digit selected is to be incremented or held constant.

```

50 2  IF (SWT$FLG AND SET$LIMIT) = SET$LIMIT THEN
51 2  DO;
52 3  CALL RESET$ALARMS;
53 3  HI$LO = SHR((SWT$FLG AND SET$HI$LO),1);
54 3  READING = LIMIT(HI$LO);
55 3  CALL BINTOBCD;
56 3  IF (SWT$FLG AND LEAVE$IT) <> LEAVE$IT THEN

```

Another counter is used to control digit incrementing. Its purpose is to control the rate at which the selected digit is to be incremented. The major loop in the program has a 4-millisecond delay. Thus, 16 A/D conversions require a period of 64 ms which provides an update frequency of 16 readings per second. This is too fast to accurately select a desired digit which is being incremented. SET\$COUNT insures eight update periods (512 ms) between each increment. After the digit has been incremented, the BCD limit value is converted back to binary to set the respective limit. This concludes the action taken when setting limits.

```

57 3  DO;
58 4  IF SET$COUNT = 0 THEN
59 4  DO;
60 5  SET$COUNT = 7;
61 5  DIGIT = SHR((SWT$FLG AND DIGIT$SLCT),2);
62 5  IF DIGIT$DATA(DIGIT) = 9 THEN
63 5  DIGIT$DATA(DIGIT) = 0;
64 5  ELSE
65 5  DIGIT$DATA(DIGIT) = DIGIT$DATA(DIGIT) + 1;
66 5  CALL BCDTOBIN(.DIGIT$DATA,.LIMIT(HI$LO));
67 4  END;
68 4  SET$COUNT = SET$COUNT - 1;
69 3  END;

```

Testing the Averaged Reading. If the set limit switch is OFF, then the averaged reading is to be tested and displayed. The averaged reading is converted to BCD and then a test is performed to determine whether the reading is to be compared with the upper and lower limits.

```

70 2  ELSE
71 3  DO;
72 3  CALL BINTOBCD;
73 3  IF (SWT$FLG AND ENABLE$ALARM) = ENABLE$ALARM THEN

```

The reading is compared with both the upper and lower limits if the alarms have been enabled. The results of the tests are used to set and reset the corresponding alarm output bits.

```

73 3  DO;
74 4  IF READING < LIMIT(0) THEN
75 4  OUTPUT(CWR) = SET$ALARM$LO;
76 4  ELSE
77 4  OUTPUT(CWR) = RESET$ALARM$LO;
78 4  IF READING > LIMIT(1) THEN
79 4  OUTPUT(CWR) = SET$ALARM$HI;
80 4  ELSE
81 4  OUTPUT(CWR) = RESET$ALARM$HI;
82 3  END;
83 2  END UPDATE;

```

If the alarms are not enabled, both the alarms are reset to the "off" condition.

```

81 3  ELSE
82 3  CALL RESET$ALARMS;
83 2  END UPDATE;

```

Main Program. The main program is shown below. Its purpose is to initialize the system and then to cycle, continuously executing the code previously described.

```

/*****
MAIN$PROGRAM:
*****/
84 1  CALL INIT;
85 1  DO FOREVER;
86 2  ACCUM$RDNG = ACCUM$RDNG + READING;
87 2  IF UPDATES$COUNT = 0 THEN
88 2  CALL UPDATE;
89 2  ELSE
90 2  UPDATES$COUNT = UPDATES$COUNT - 1;
91 2  CALL DISPLAY$UPDATE;
92 2  CALL TIME(40);
93 2  READING = ATOD;
94 1  END;

```

SUMMARY/CONCLUSIONS

The goal of this application example is to demonstrate some of the common functions required for process control systems. Rather than show a small portion of a larger, more complex problem, this example was chosen because it presents a complete solution to a smaller problem. In summary, refreshing a multiplexed display was shown. Conversion procedures for binary to BCD and BCD to binary were used. A simple technique, in terms of hardware requirements, was used to enter lower and upper test values. And, limits testing was done, providing alarm indicators.

The reading is compared with both the upper and lower limits if the alarm have been enabled. The results of the tests are used to set and reset the corresponding alarm.

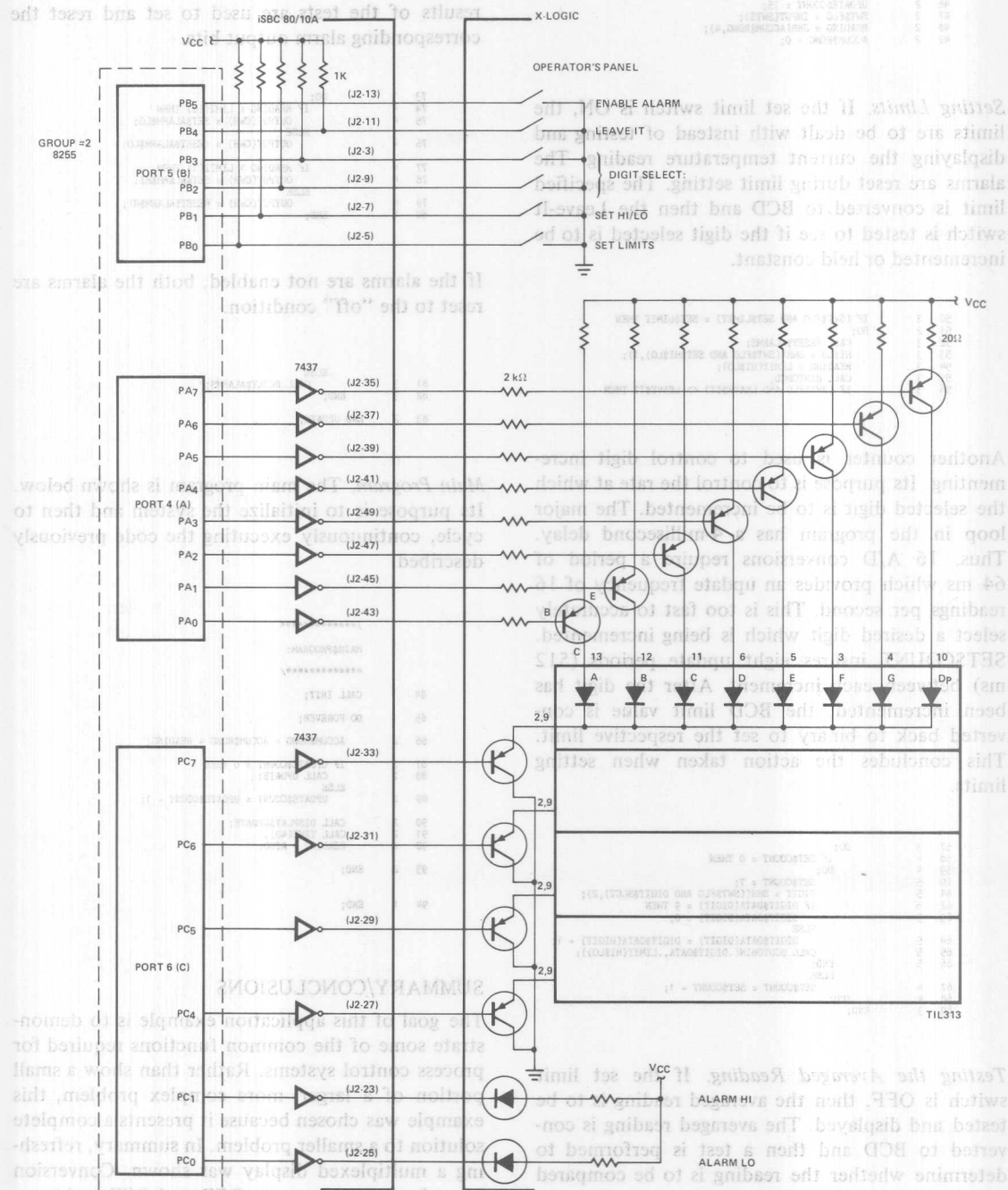


Figure 9. Operator's Panel Schematic

I/O DEVICE CONTROLLER

Peripheral processors have become common elements in computer systems of all sizes and capabilities. The purpose of such a processor is to relieve a central processor from the burden of handling I/O devices. In effect, it is a form of distributed processing. The iSBC 80/10A can be used as a peripheral processor and/or as an I/O device controller. In such a capacity it can significantly reduce the amount of hardware required to interface peripherals. Because the iSBC 80/10A controls only I/O, it is of little consequence that it must do a great deal of detail work that otherwise wastes the processing capability of a larger central processor.

Consider the activity of producing a listing on a line printer. The overhead in maintaining a program in the queue of a central processor which is producing data for a line printer can seriously impact system throughput. If, however, the program were to send the list to a disk file and then command a peripheral processor to take care of the printing, a significant improvement in system performance may be achieved. Printers represent one example of a large number of I/O devices that can be controlled by an iSBC 80/10A. Other devices include cassettes, magnetic tape drives, paper tape readers and punches, etc.

Character Printer Controller Application Example

The control of a Centronics 306 character printer is used as an I/O device controller application example. This example shows the interrupt capability of mode 1 8255 operation. A block diagram of the printer controller is shown in Figure 10 and a schematic in Figure 11.

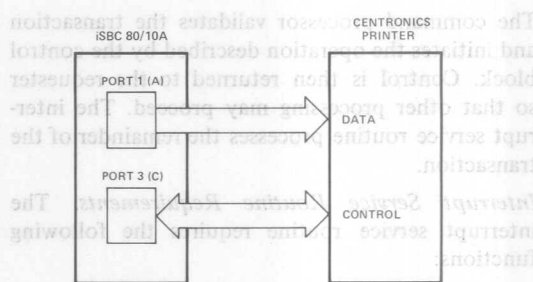


Figure 10. Printer Controller Block Diagram

When the mode 1 or mode 2 configuration is used, software is generally required to support interrupts used in conjunction with handshaking operations. Software routines written for an interrupt driven environment tend to be more complex than status driven routines. The added complexity is because interrupt-driven systems are constructed such that other software tasks are run while the I/O transaction is in progress. A software routine that controls a peripheral device is generally referred to as a device driver. One method of implementing an interrupt-driven device driver is to partition the device driver into a "command processor" and an "interrupt service routine." The command processor is the module that validates and initiates user program requests to the device driver. A common method of passing information between the various software programs is to have the requesting routine provide a device control block in memory. The device control block used in this application example is shown in Table 2.

Table 2. Printer Software Control Block

NAME	POSITION	DEFINITION
Status	Byte 0	A 1-byte field which defines the completion status of an I/O. 00 = Good completion 01 = Error — command already in progress.
Buffer Address	Byte 1, 2	Pointer to the start of the data to print.
Character Count	Byte 3	Count of the number of characters to print.
Character Transferred Count	Byte 4	The number of characters transferred.
Completion Address	Byte 5, 6	Address of a user supplied routine which will be called after the I/O has been performed.

The command processor validates the transaction and initiates the operation described by the control block. Control is then returned to the requester so that other processing may proceed. The interrupt service routine processes the remainder of the transaction.

Interrupt Service Routine Requirements. The interrupt service routine requires the following functions:

1. The state of the machine (registers, status, etc.) must be saved so that it may be restored after the interrupt is processed.
2. The source of the interrupt must be determined. The hardware may support a register which indicates the interrupting device, or the software may poll the device status registers.
3. Data must be passed to or from the device.
4. Control must be passed to the requesting routine at the completion of the I/O.
5. The state of the machine must be restored before returning to the interrupted program.

Printer Controller Program. The software for this application has been coded using Intel® 8080 Macro Assembly Language.

```

0:
1: *****
2:
3: I/O DEVICE CONTROLLER APPLICATION
4:
5:
6: INTERRUPT DRIVEN
7: CHARACTER PRINTER
8: *****

```

The following program equates are used to allow symbolic reference to the 8255 ports. Group #1 8255 on the iSBC 80/10A has been used because it will support mode 1 operation.

```

10:
11: *****
12: PROGRAM EQUATES
13: *****
14: PORTA EQU 0E4H ; 8255 PORT A
15: PORTB EQU 0E5H ; 8255 PORT B
16: PORTC EQU 0E6H ; 8255 PORT C
17: CWR EQU 0E7H ; 8255 CONTROL WORD REGISTER

```

An initialization control word sent to the control word register of the 8255 will set up the desired configuration.

```

18:
19: *****
20: INITIALIZATION CONTROL WORD
21:
22: USED TO CONFIGURE THE 8255 AS FOLLOWS:
23:
24:
25: PORT A - OUTPUT MODE 1
26: PORT B - INPUT MODE 0 (NOT USED)
27: PORT C LOWER - OUTPUT
28:
29: *****
30: ICW EQU 10101010B ; INITIALIZATION CONTROL WORD
31: *****

```

The bit set/reset capability of the 8255 is used to control the strobe to the printer and to enable/disable interrupts from the 8255.

```

32: SET/RESET CONTROL WORDS
33: *****
34: STBON EQU 00000001B ; SET STROBE
35: STBOF EQU 00000000B ; RESET STROBE
36: *****
37: 8255 ENABLE/DISABLE INTERRUPT CONTROL WORDS
38: *****
39: IEN EQU 00001101B ; ENABLE INTERRUPTS
40: IDN EQU 00001100B ; DISABLE INTERRUPTS
41: *****

```

Device status, control block, and completion equates are shown below.

```

42: DEVICE STATUS EQUATES
43: *****
44: LPSBY EQU 0080H ; BUFFER FULL (LINE PRINTER BUSY)
45: INTRA EQU 008H ; INTERRUPT REQUEST
46: *****
47: CONTROL BLOCK EQUATES
48: *****
49: CBST EQU 000H ; STATUS BYTE
50: CBUF EQU 01H ; BUFFER ADDRESS
51: CBCC EQU 03H ; CHARACTER COUNT
52: CBCT EQU 04H ; CHARACTER TRANSFERRED COUNT
53: CBCHP EQU 05H ; COMPLETION SERVICE ADDRESS
54: *****
55: COMPLETION STATUS EQUATES
56: *****
57: STGD EQU 00H ; GOOD COMPLETION
58: STETD EQU 01H ; ERROR - COMMAND ALREADY IN PROGRESS
59: *****

```

There are two origin statements in this program. The first origin at 38 hexadecimal is the entry point used when an interrupt is generated by the 8255. A jump instruction to the printer interrupt routine is stored at that location. The second origin at 3000 hexadecimal is the address where the rest of the code will be located.

```

60: RESTART 7 ENTRY POINT
61: *****
62: ORG 003BH
63: JMP PINT
64: *****
65: PROGRAM ORIGIN
66: *****
67: ORG 3000H
68: *****

```

An initialization subroutine issues the mode control word to the 8255 control word register after reset of the device. The printer strobe must then be disabled.

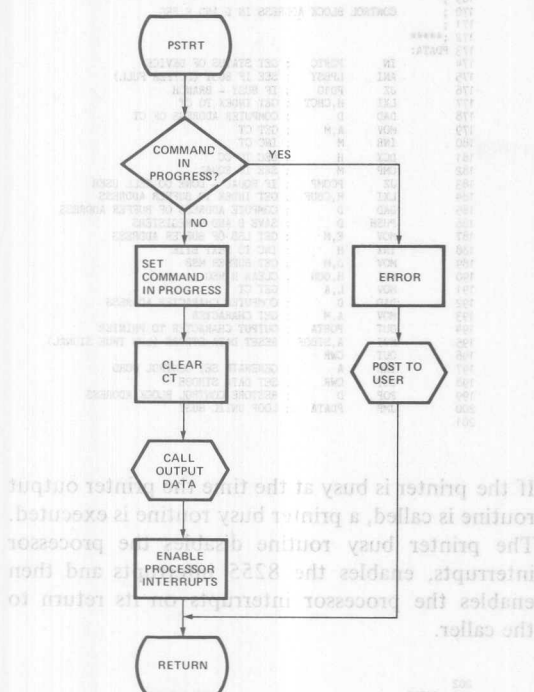
```

69: INITIALIZATION ROUTINE
70:
71: A,H,L REGISTERS MODIFIED
72:
73:
74: *****
75: INIT:
76: MVI A,ICW ; GET MODE CONTROL WORD
77: OUT CWR ; OUTPUT TO CONTROL WORD REGISTER
78: MVI A,STBON ; GET SET DATA STROBE CONTROL WORD
79: OUT CWR ; SET DATA STROBE (LOW TRUE SIGNAL)
80: RET ; RETURN TO CALLER
81: *****

```

The command processor is started by the user routine through a subroutine call to PSTRT, with the address of the control block in the D and E registers. The command processor insures that an I/O operation is not already in progress, starts the I/O, enables interrupts, and returns to the caller so that other processing may proceed.

The flowchart and listing for the command processor are shown below.



```

82 *****
83 *****
84 *****
85 *****
86 *****
87 *****
88 *****
89 *****
90 *****
91 *****
92 *****
93 *****
94 *****
95 *****
96 *****
97 *****
98 *****
99 *****
100 *****
101 *****
102 *****
103 *****
104 *****
105 *****
106 *****
107 *****
108 *****
109 *****
110 *****
111 *****
112 *****
113 *****
114 *****
115 *****

```

COMMAND PROCESSOR

INPUTS: CONTROL BLOCK ADDRESS IN D AND E REGISTERS

OUTPUTS: START I/O OR ERROR STATUS IN CONTROL BLOCK

A, H, L REGISTERS MODIFIED

```

PSTRT:
LDA PIPRG-1 ; GET PRINT IN PROGRESS BLOCK ADDRESS
ANA A ; SEE IF ZERO (PRINT IN PROGRESS)
; IF BLOCK ADDRESS NOT EQUAL TO ZERO THEN
; PRINT IN PROGRESS
; IF YES - BRANCH TO ERROR
JNZ PSTE
XCHG PIPRG ; SAVE CONTROL BLOCK ADDRESS
LXI H,CBCT ; GET INDEX TO CT
DAD D ; COMPUTE ADDRESS OF CT
MVI M,00H ; CLEAR CT
CALL PDATA ; START I/O
EI ; ENABLE PROCESSOR INTERRUPTS
RET ; RETURN TO CALLER

```

ERROR - TRANSACTION ALREADY IN PROGRESS

```

PSTE:
MVI A,STE1 ; GET ERROR STATUS CODE
JMP POST ; PASS CONTROL TO COMPLETION ROUTINE

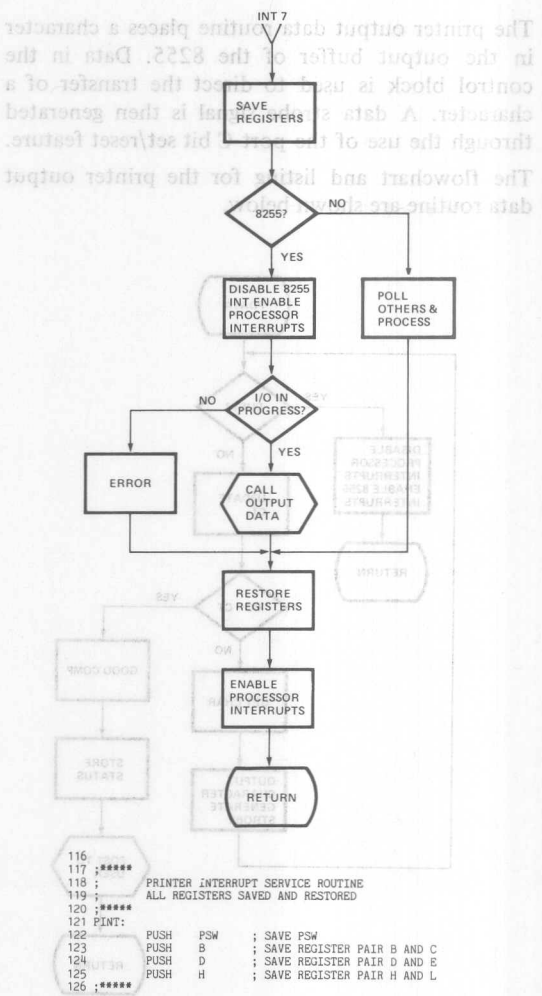
```

Interrupt Processing. When the 8255 generates an interrupt, the interrupt request is serviced by the 8080A CPU. The CPU disables processor interrupts and then executes the instruction at location 38 hexadecimal, which is a jump to the interrupt service routine. The interrupt service routine saves the processor state and polls the 8255 to determine the source of the interrupt. Once the interrupting device is identified, the printer output data routine

is called. After the entire buffer has been printed, the interrupt service routine passes control to the user-supplied completion routine. Before returning from the interrupt, the state of the processor is restored.

There are a number of error conditions which may occur, such as an interrupt from a device that does not have an active control block, or an interrupt when polling establishes that no device requires service. Neither of these errors should occur, but if they do, the driver should perform in a consistent fashion. The recovery routines implemented to handle these interrupt error conditions are determined by the environment of the particular application.

The flowchart and listing for the printer interrupt service routine are shown below.



```

116 *****
117 *****
118 *****
119 *****
120 *****
121 *****
122 *****
123 *****
124 *****
125 *****
126 *****

```

PRINTER INTERRUPT SERVICE ROUTINE
ALL REGISTERS SAVED AND RESTORED

```

PUSH PSW ; SAVE PSW
PUSH B ; SAVE REGISTER PAIR B AND C
PUSH D ; SAVE REGISTER PAIR D AND E
PUSH H ; SAVE REGISTER PAIR H AND L

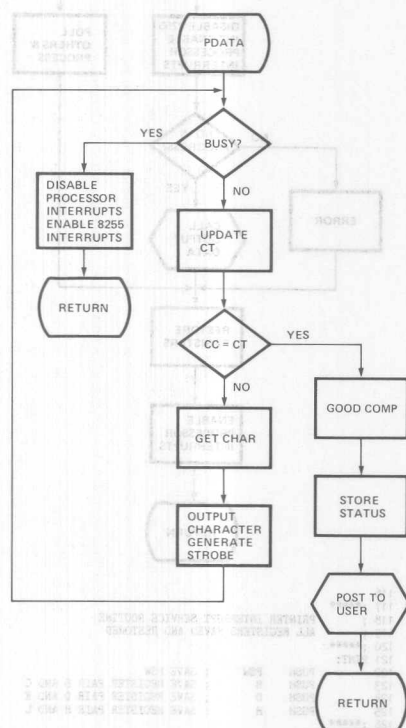
```

```

127 ; POLL INTERRUPT SOURCE - SEE OF 8255
128 ;*****
129 IN PORTC ; GET STATUS OF DEVICE
130 ANI INTRA ; SEE IF INT
131 JZ PPOLL ; NO - BRANCH TO POLL OTHER DEVICES IF ANY
132 MVI A, IDN ; GET 8255 INT DISABLE CONTROL WORD
133 OUT CWR ; DISABLE DEVICE INTERRUPTS
134 EI ; ENABLE PROCESSOR INTERRUPTS
135 LHLD PIPRG ; GET CONTROL BLOCK ADDRESS
136 XRA A ; CLEAR A REG
137 CMP H ; SEE IF PRINT IN PROGRESS
138 JZ PIER1 ; NO - BRANCH TO ERROR ROUTINE
139 CALL PDATA ; PRINT DATA
140 ;*****
141 ; RESTORE REGISTERS AND RETURN FROM INTERRUPT
142 ;*****
143 PRIN:
144 POP H ; RESTORE REGISTER PAIR H AND L
145 POP D ; RESTORE REGISTER PAIR D AND E
146 POP B ; RESTORE REGISTER PAIR B AND C
147 POP PSW ; RESTORE PSW AND A
148 EI ; ENABLE PROCESSOR INTERRUPTS
149 RET ; RETURN TO INTERRUPTED PROCESS
150 ;*****
151 ; POLL OTHER DEVICES IF ANY
152 ; IF NO OTHER DEVICES TO POLL - USER SUPPLIED ERROR
153 ; RECOVERY ROUTINE.
154 ;*****
155 PPOLL:
156 JMP PRIN ; RETURN
157 ;*****
158 ERROR - INTERRUPT FROM IDLE DEVICE
159 ; USER SUPPLIED ERROR RECOVERY ROUTINE
160 ;*****
161 PIER1:
162 JMP PRIN ; RETURN
163
164

```

The printer output data routine places a character in the output buffer of the 8255. Data in the control block is used to direct the transfer of a character. A data strobe signal is then generated through the use of the port C bit set/reset feature. The flowchart and listing for the printer output data routine are shown below.



```

165
166 ;*****
167 ; PRINTER OUTPUT DATA ROUTINE
168 ;*****
169 ; CONTROL BLOCK ADDRESS IN D AND E REG
170 ;*****
171 PDATA:
172 IN PORTC ; GET STATUS OF DEVICE
173 ANI LPSBY ; SEE IF BUSY (BUFFER FULL)
174 JZ PD10 ; IF BUSY - BRANCH
175 LXI H, CBCT ; GET INDEX TO CT
176 DAD D ; COMPUTER ADDRESS OF CT
177 MOV A, M ; GET CT
178 INR M ; INC CT
179 CMP H ; DEC TO CC
180 JZ PCOMP ; SEE IF EQUAL
181 PCOMP:
182 IF EQUAL - DONE GO TELL USER
183 LXI H, CBUP ; GET INDEX TO BUFFER ADDRESS
184 DAD D ; COMPUTE ADDRESS OF BUFFER ADDRESS
185 PUSH D ; SAVE D AND E REGISTERS
186 MOV E, M ; GET LSB OF BUFFER ADDRESS
187 INC E ; INC TO NEXT BYTE
188 MOV D, M ; GET BUFFER MSB
189 MVI H, 00H ; CLEAR H REG
190 MOV L, A ; GET CT
191 DAD D ; COMPUTER CHARACTER ADDRESS
192 MOV A, M ; GET CHARACTER
193 OUT PORTA ; OUTPUT CHARACTER TO PRINTER
194 MVI A, STBOP ; RESET DATA-STROBE (LOW TRUE SIGNAL)
195 OUT CWR ; RESET DATA-STROBE
196 INR A ; GENERATE SET CONTROL WORD
197 OUT CWR ; SET DATA STROBE
198 POP D ; RESTORE CONTROL BLOCK ADDRESS
199 JMP PDATA ; LOOP UNTIL BUSY
200
201

```

If the printer is busy at the time the printer output routine is called, a printer busy routine is executed. The printer busy routine disables the processor interrupts, enables the 8255 interrupts and then enables the processor interrupts on its return to the caller.

```

202 ;*****
203 ; PRINTER BUSY - RETURN
204 ;*****
205 PD10:
206 DI ; DISABLE INTERRUPTS
207 MVI A, IDN ; ENABLE DEVICE INTERRUPTS
208 OUT CWR ; SET INTERRUPT ENABLE
209 RET ; RETURN TO CALLER
210 ;*****
211
212 ; POST GOOD COMPLETION TO USER
213 ;*****
214 PCOMP:
215 MVI A, STGD ; GET GOOD STATUS CODE
216 CALL POST ; POST TO USER
217 XRA A ; CLEAR A REG
218 STA PIPRG+1 ; CLEAR COMMAND IN PROGRESS ADDRESS
219 RET ; RETURN TO CALLER
220 ;*****
221
222 ; POST TO USER COMPLETION ROUTINE
223 ;*****
224
225 INPUTS: STATUS CODE IN A REG
226 CONTROL BLOCK ADDRESS IN D AND E REG
227
228 OUTPUTS: PASSES CONTROL TO USER COMPLETION ADDRESS
229 SPECIFIED IN CONTROL BLOCK
230 WITH CONTROL BLOCK ADDRESS IN D AND E REG
231
232 A, H, L, B, C REG MODIFIED
233 ;*****
234

```

The post to user completion routine obtains the completion address from the device control block and passes control to the user routine.

```

235 POST:
236       XCHG     MOV     M,A      ; UPDATE STATUS
237
238       XCHG     LXI     H,CBOMP  ; GET INDEX TO COMPLETION ADDRESS
239
240       DAD      D          ; COMPUTE ADDRESS
241       MOV      C,M        ; GET LSB OF COMPLETION ADDRESS
242
243       INX      H          ; INC TO NEXT BYTE
244       MOV      B,M        ; GET MSB OF COMPLETION ADDRESS
245       PUSH     B          ; PUSH ADDRESS ON STACK
246       RET       ; PASS CONTROL TO USER ROUTINE
247
248 ***** DATA AND TABLES
249
250 PIPRG: DW      3D00H      ; IN PROGRESS CONTROL BLOCK ADDRESS
251
252 ***** ; IF DATA = 0 NO CONTROL BLOCK IN PROGRESS
253 ***** ; IF DATA <> 0 CONTROL BLOCK IN PROGRESS
254 ***** END OF MODE ONE EXAMPLE
255 *****
256       END

```

SUMMARY/CONCLUSIONS

The iSBC 80/10A has the capability to function in the capacity of a peripheral processor and/or a device controller. This capability is provided in part by the interrupt support logic accompanying the parallel I/O ports. This application example shows how the iSBC 80/10A requires only an interconnect to the device to be controlled.

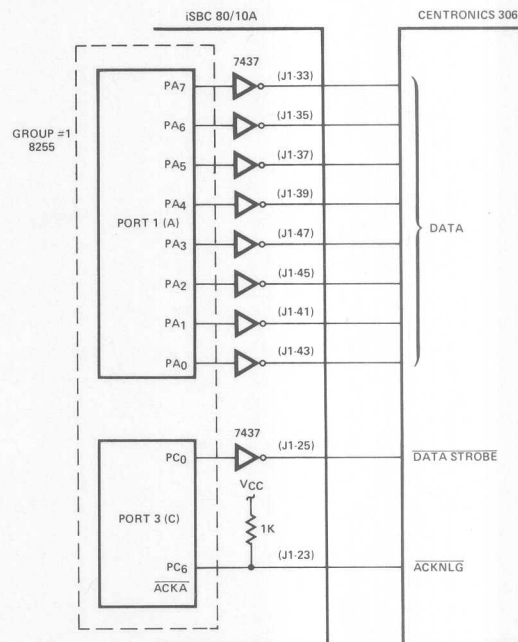


Figure 11. Printer Controller Schematic

CONCLUSION

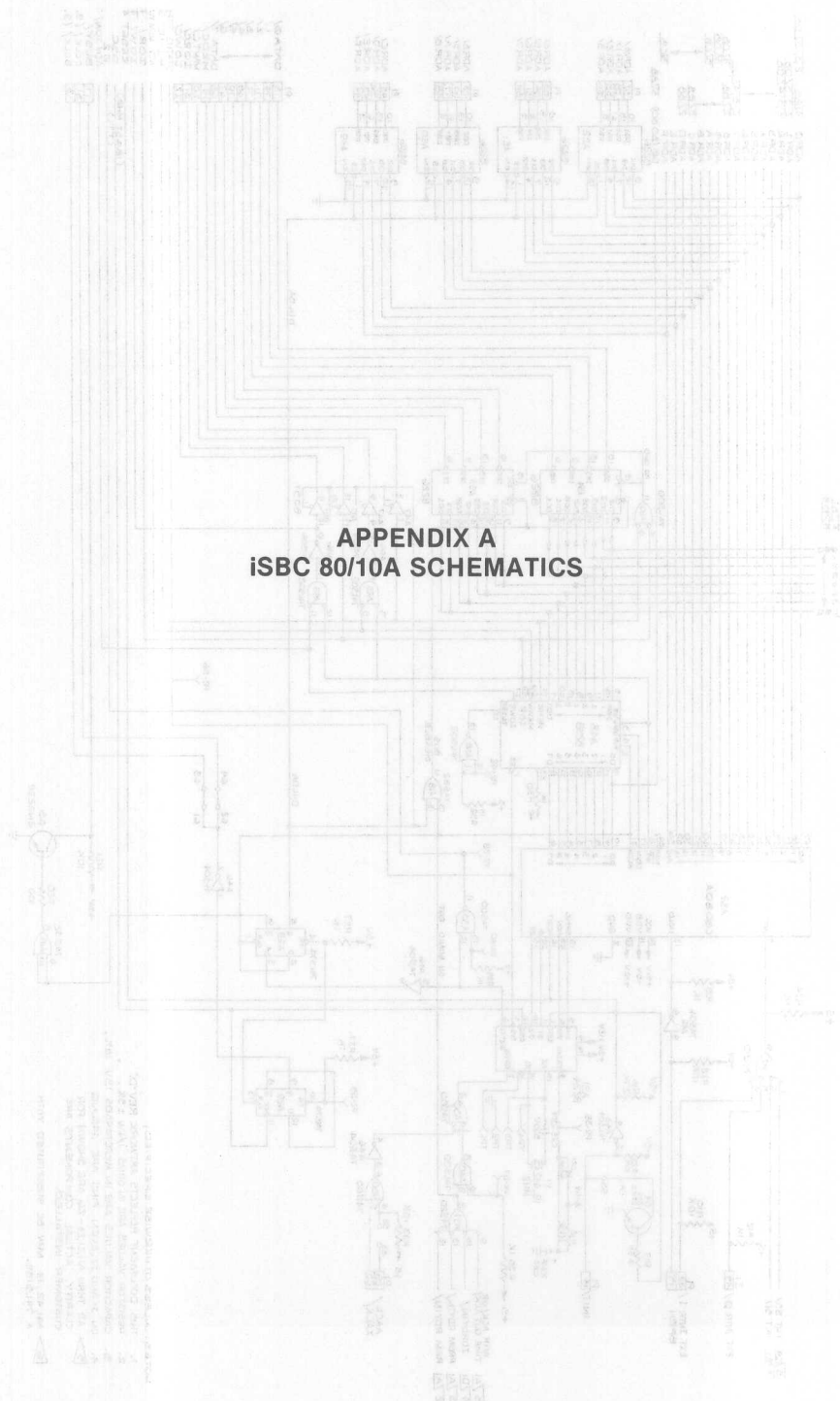
The purpose of this application note has been to expose the reader to a broad spectrum of potential applications of the Intel iSBC 80/10A and System 80/10 products. Applications have been presented in the areas of instrumentation, communication, process control and I/O device control. The examples were limited to short problems that could be completely described.

Intel's PL/M-80 and 8080 Macro Assembly Language were both used in the examples. Instead of using only assembly language, it was felt that PL/M-80 should also be shown. Coding in an algorithmic language is generally more natural than assembly language and provides these added benefits: reduced program development time and cost, improved product reliability, and easier program maintenance.

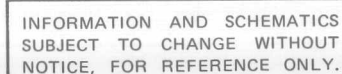
While the task of actually configuring the SBC 80/10 for the applications has not been described in this note, detailed instructions are contained in the tables of Chapter 4 in the *iSBC 80/10 and iSBC 80/10A Single Board Computer Hardware Reference Manual*.

The Intel iSBC 80/10A has been designed to provide users with subsystems that have processing power, memory storage, parallel and serial programmable I/O interfaces. A design goal of the iSBC 80/10A was to minimize the requirements for customized interface hardware in user applications. This application note has demonstrated the achievement of this goal. The net effect is to reduce the number of tedious design tasks, thus allowing the systems designer to concentrate on systems integration and other problems unique to his job.

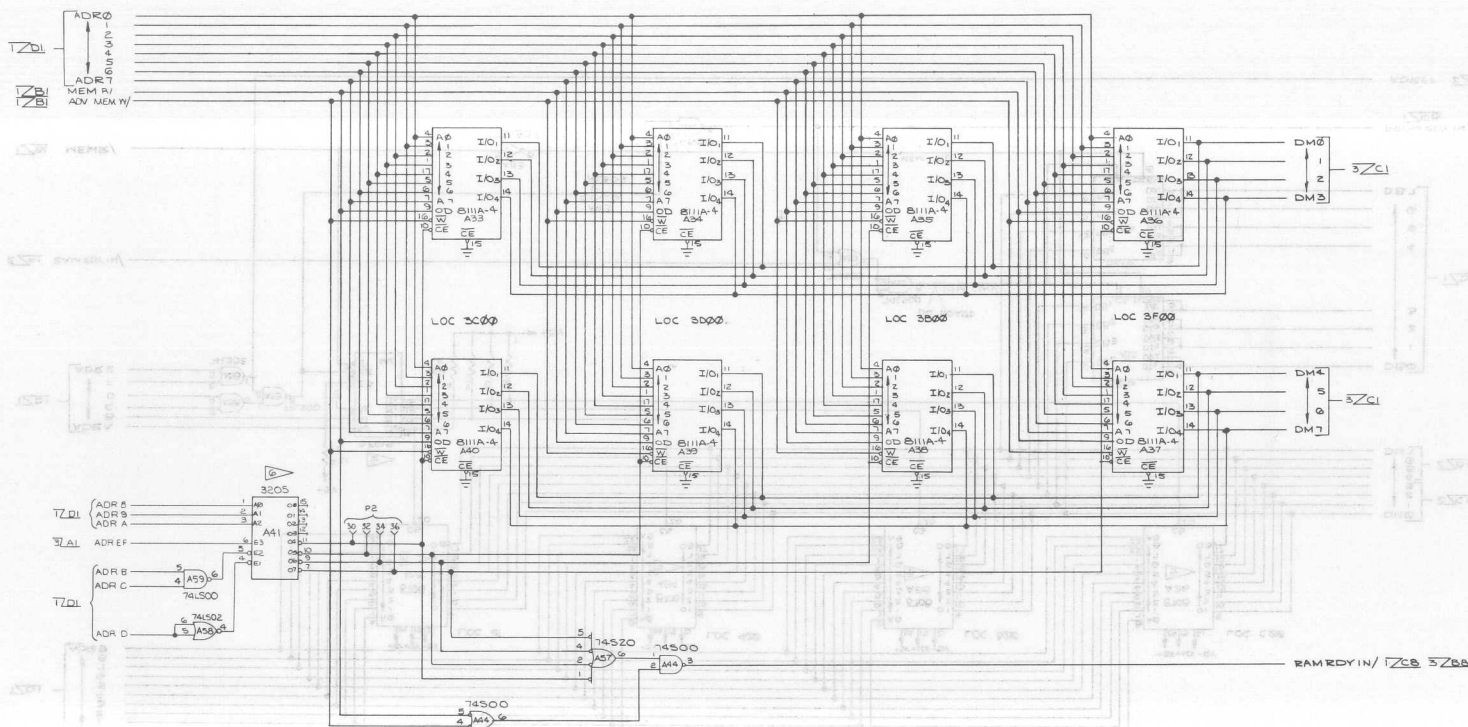
NOTICE FOR REFERENCE (1)
 SUBJECT TO CHANGE MIT
 INFORMATION AND 3-16-61



APPENDIX A ISBC 80/10A SCHEMATICS

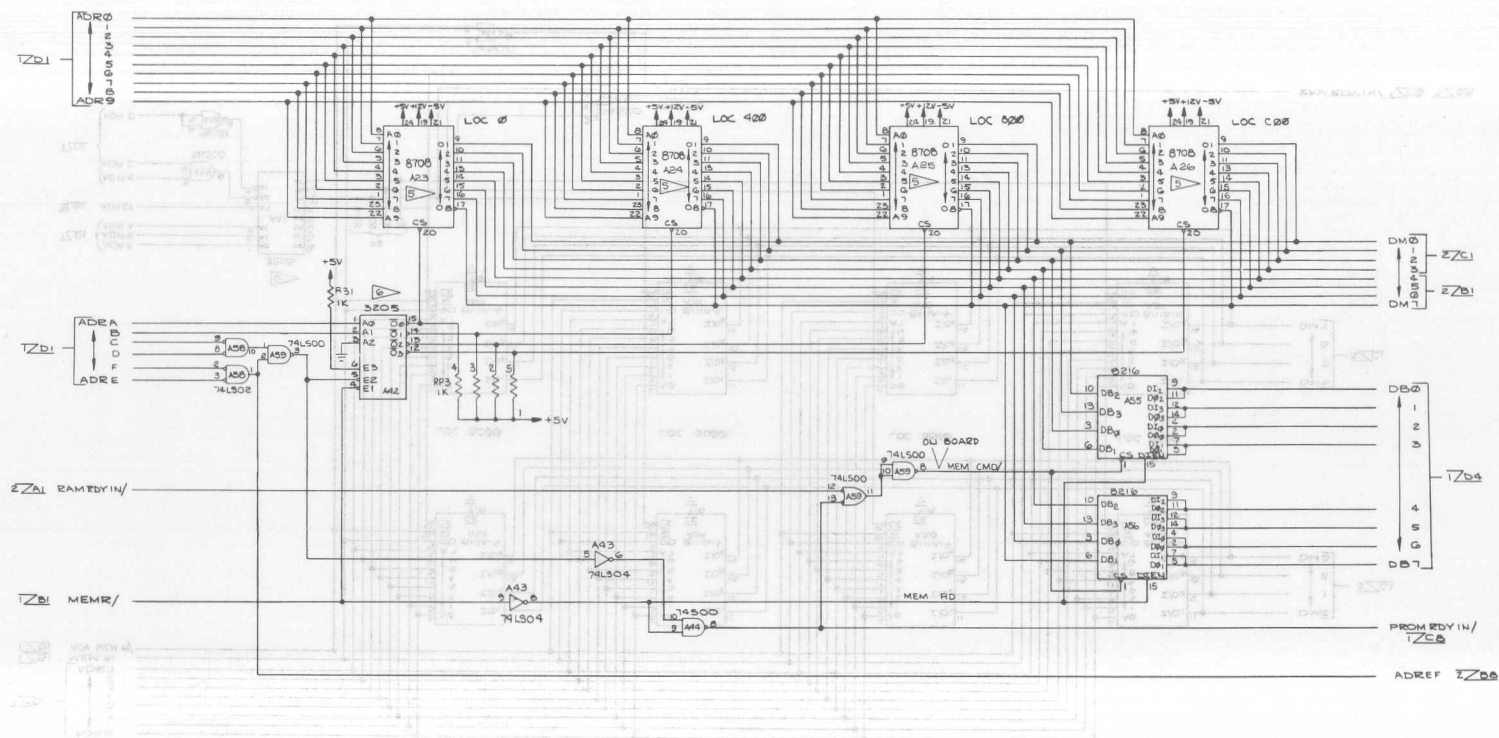


NOTICE FOR REFERENCE ONLY
SUBJECT TO CHANGE WITHOUT
NOTICE AND SCHEMATICS

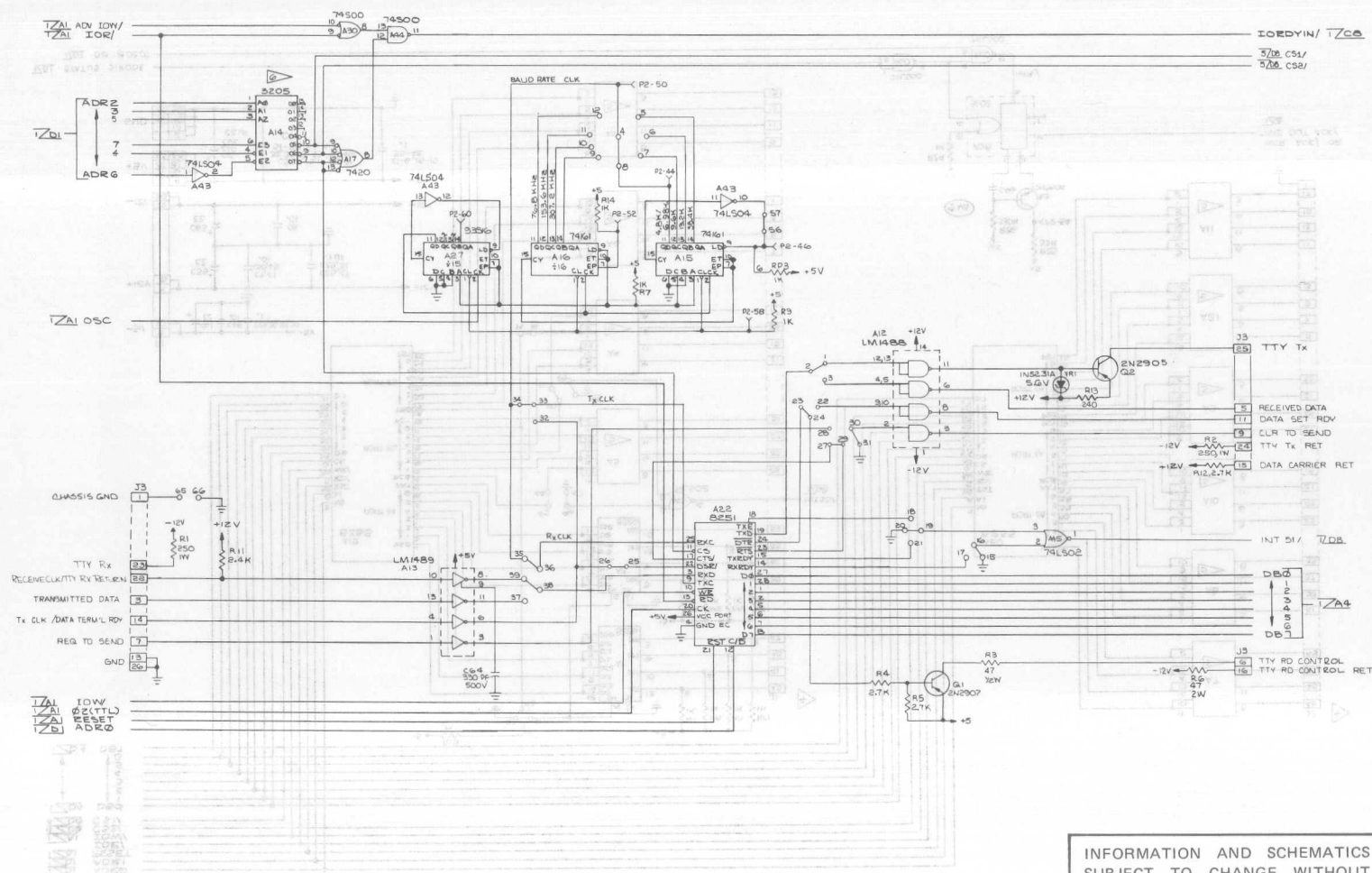


INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.

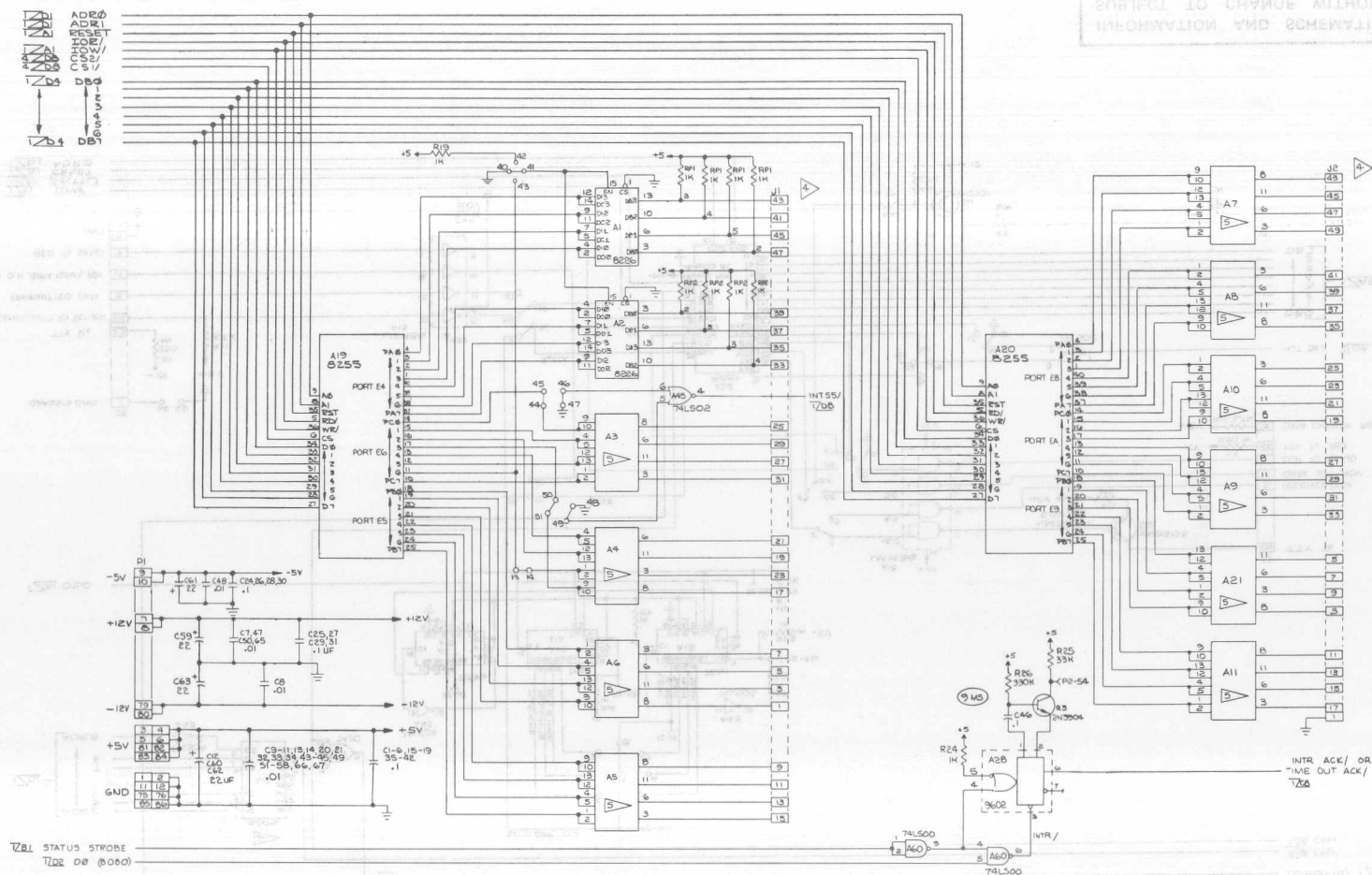
INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.



INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.

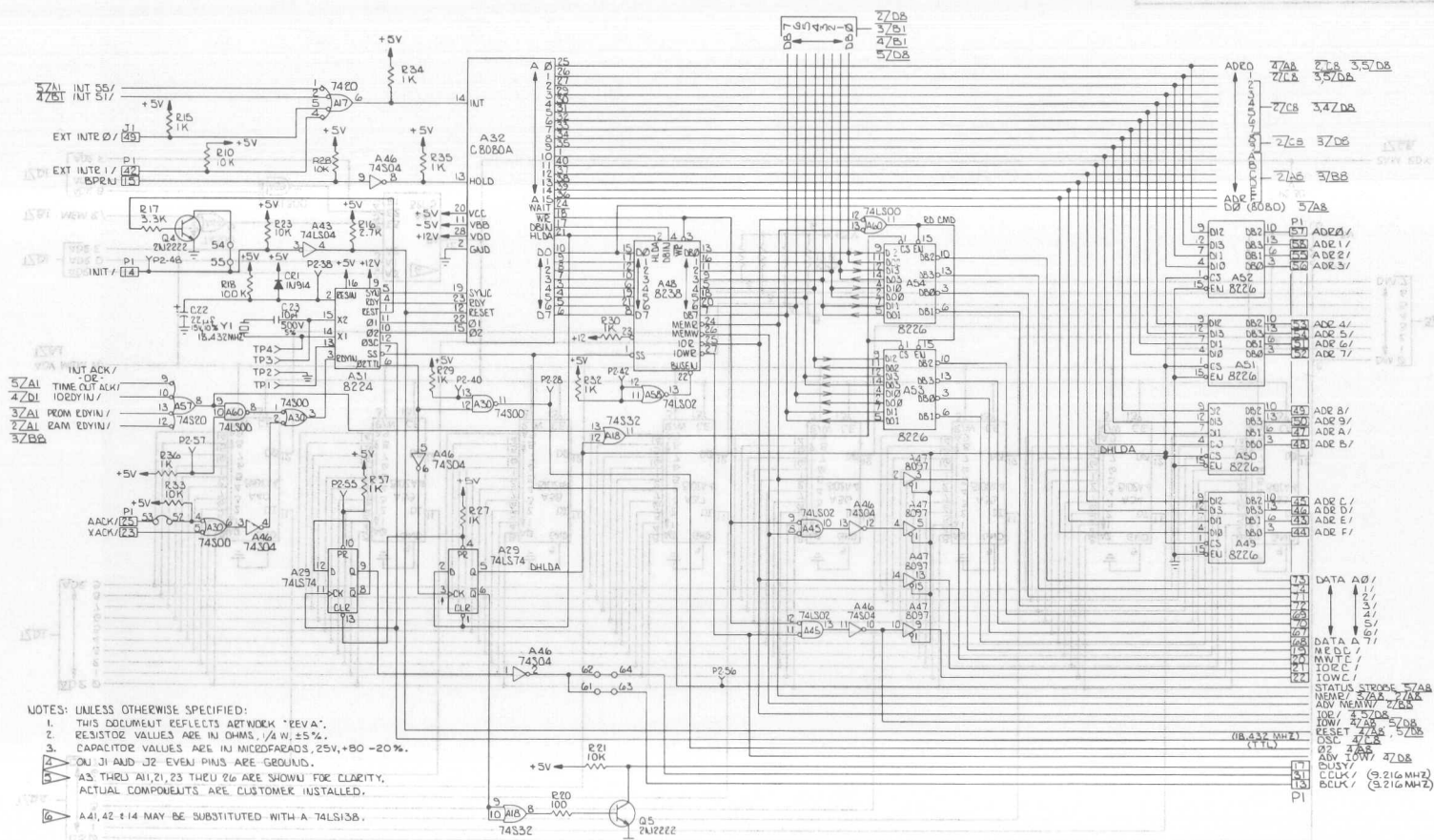


NOTICE: FOR REFERENCE ONLY.
SUBJECT TO CHANGE WITHOUT
NOTICE AND SCHEMATICS

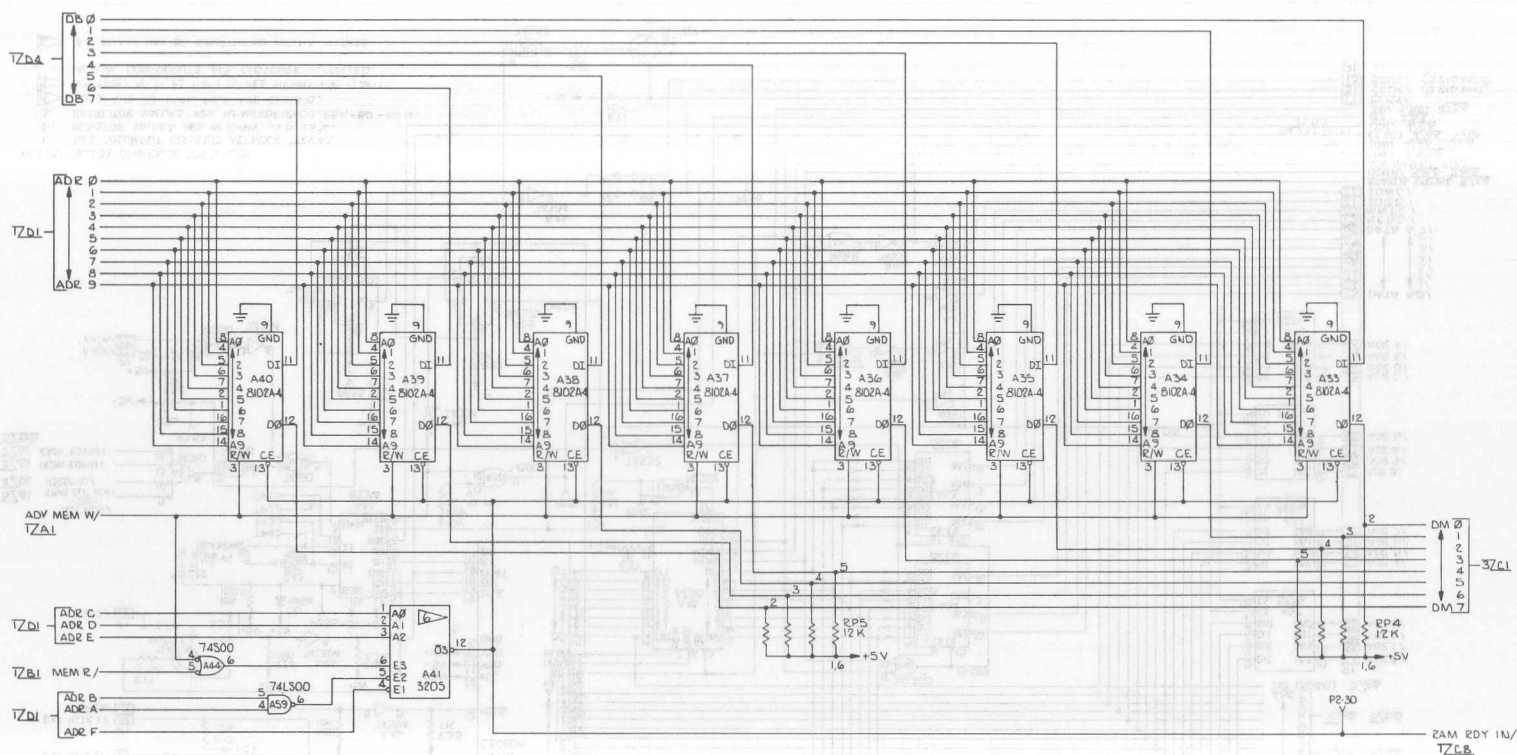


INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.

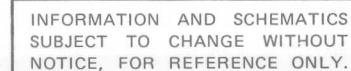
INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.



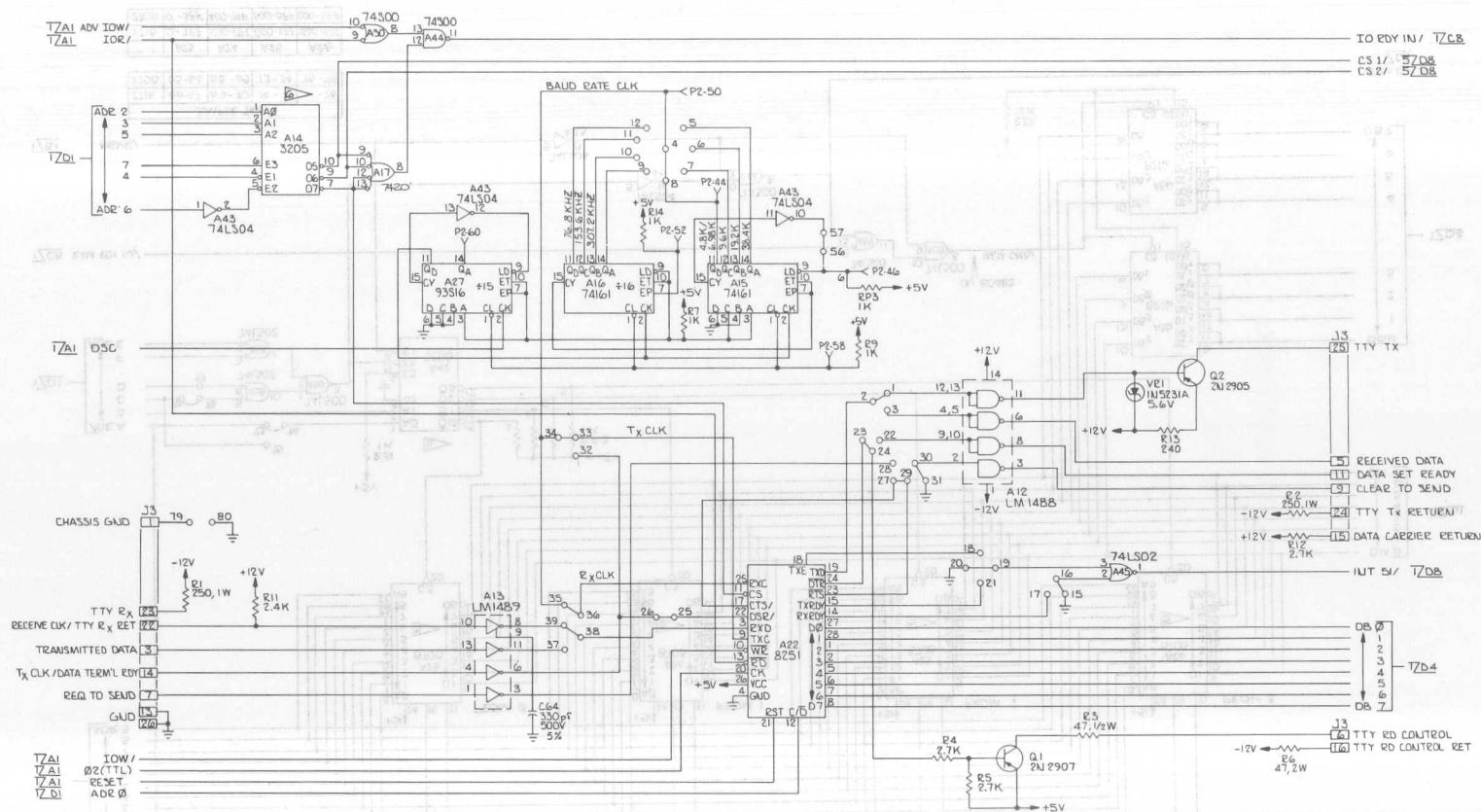
NOTICE FOR REFERENCE ONLY
SUBJECT TO CHANGE WITHOUT
NOTICE AND SCHEDULE



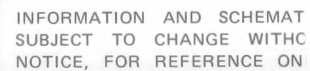
INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.



NOTICE: FOR REFERENCE ONLY
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY



INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.





APPLICATION NOTE ADDENDUM

AP-28A

July 1980

Intel MULTIBUS™ Interfacing

Joe Barthmaier
OEM Microcomputer
Systems Applications

AFN-01931A

Intel® MULTIBUS™ Interfacing

Contents

I. INTRODUCTION	1-47
II. MULTIBUS™ SYSTEM BUS DESCRIPTION	1-47
Overview	1-47
MULTIBUS™ Signal Descriptions	1-47
Operating Characteristics	1-51
MULTIBUS™ Slave Interface Circuit Elements	1-60
III. MULTIBUS™ SLAVE DESIGN EXAMPLE	1-62
Functional/Programming Characteristics	1-62
Theory of Operation	1-63
IV. SUMMARY	1-66
APPENDIX A — MULTIBUS™ PIN ASSIGNMENTS	1-67
APPENDIX B — BUS TIMING SPECIFICATIONS	1-69
APPENDIX C — BUS DRIVERS, RECEIVERS, AND TERMINATIONS	1-71
APPENDIX D — BUS POWER SUPPLY SPECIFICATIONS	1-73
APPENDIX E — MECHANICAL SPECIFICATIONS	1-74
APPENDIX F — MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC 8/16-BIT VERSION	1-75
APPENDIX G — MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC 8-BIT VERSION	1-77

of the Intel OEM Computer Product Line can be attributed to the design of the Intel MULTIBUS system bus. The bus structure provides a common element for communication between a wide variety of system modules which include: Single Board Computers, memory, digital, and analog I/O expansion boards, and peripheral controllers.

The purpose of this application note is to help you develop a working knowledge of the Intel MULTIBUS specification. This knowledge is essential for configuring a system containing multiple modules. Another purpose is to provide you with the information necessary to design a bus interface for a slave module. One of the tools that will be used to achieve this goal is the complete description of a MULTIBUS slave design example. Other portions of this application note provide an in depth examination of the bus signals, operating characteristics, and bus interface circuits.

This application note was originally written in 1977. Since 1977, the MULTIBUS specification has been significantly expanded to cover operation with both 8 and 16-bit system modules and with an auxiliary power bus. This application note now contains information on these new MULTIBUS specification features.

In addition, a detailed MULTIBUS specification has also been published which provides the user with further information concerning MULTIBUS interfacing. The MULTIBUS specification and other useful documents are listed in the overleaf of this note under Related Intel Publications.

II. MULTIBUS™ SYSTEM BUS DESCRIPTION

Overview

The Intel MULTIBUS signal lines can be grouped in the following categories: 20 address lines, 16 bidirectional data lines, 8 multilevel interrupt lines, and several bus control, timing and power supply lines. The address and data lines are driven by three-state devices, while the interrupt and some other control lines are open-collector driven.

Modules that use the MULTIBUS system bus have a master-slave relationship. A bus master module can drive the command and address lines: it can control the bus. A Single Board Computer is an example of a bus master. A bus slave cannot

bus masters and slaves.

Notice that a system may have a number of bus masters. Bus arbitration results when more than one master requests control of the bus at the same time. A bus clock is usually provided by one of the bus masters and may be derived independently from the processor clock. The bus clock provides a timing reference for resolving bus contention among multiple requests from bus masters. For example, a processor and a DMA (direct memory access) module may both request control of the bus. This feature allows different speed masters to share resources on the same bus. Actual transfers via the bus, however, proceed asynchronously with respect to the bus clock. Thus, the transfer speed is dependent on the transmitting and receiving devices only. The bus design prevents slow master modules from being handicapped in their attempts to gain control of the bus, but does not restrict the speed at which faster modules can transfer data via the same bus. Once a bus request is granted, single or multiple read/write transfers can proceed. The most obvious applications for the master-slave capabilities of the bus are multiprocessor configurations and high-speed direct-memory-access (DMA) operations. However, the master-slave capabilities of the bus are by no means limited to these two applications.

MULTIBUS™ Signal Descriptions

This section defines the signal lines that comprise the Intel MULTIBUS system bus. These signals are contained on either the P1 or P2 connector of boards compatible with the MULTIBUS specification. The P1 signal lines contain the address, data, bus control, bus exchange, interrupt and power supply lines. The P2 signal lines contain the optional auxiliary signal lines. Most signals on the bus are active-low. For example, a low level on a control signal on the bus indicates active, while a low level on an address or data signal on the bus represents logic "1" value.

NOTE

In this application note, a signal will be designated active-low by placing a slash (/) after the mnemonic for the signal.

Appendix A contains a pin assignment list of the following signals:

MULTIBUS P1 Signal Lines —

Initialization Signal Line

INIT/

Initialization signal; resets the entire system to a known internal state. INIT/ may be driven by one of the bus masters or by an external source such as a front panel reset switch.

Address and Inhibit Lines

ADR0/ - ADR13/

20 address lines; used to transmit the address of the memory location or I/O port to be accessed. The lines are labeled ADR0/ through ADR9/, ADRA/ through ADRF/ and ADR10/ through ADR13/. ADR13/ is the most significant bit. 8-bit masters use 16 address lines (ADR0/ - ADRF/) for memory addressing and 8 address lines (ADR0/ - ADR7/) for I/O port selection. 16-bit masters use all twenty address lines for memory addressing and 12 address lines (ADR0/ - ADRB/) for I/O port selection. Thus, 8-bit masters may address 64K bytes of memory and 256 I/O devices while 16-bit masters may address 1 megabyte of memory and 4096 I/O devices. (The 8086 CPU actually permits 16 address bits to be used to specify I/O devices, the MULTIBUS specification, however, states that only the low order 12 address bits can be used to specify I/O ports.) In a 16-bit system, the ADR0/ line is used to indicate whether a low (even) byte or a high (odd) byte of memory or I/O space is being accessed in a word oriented memory or I/O device.

BHEN/

Byte High Enable; the address control line which is used to specify that data will be transferred on the high byte (DAT8/ - DATF/) of the MULTIBUS data lines. With current iSBC boards, this signal effectively specifies that a word (two byte) transfer is to be performed. This signal is used only in systems which incorporate sixteen bit memory or I/O modules.

INH1/

Inhibit RAM signal; prevents RAM memory devices from responding to the memory address on the system address bus. INH1/ effectively allows ROM memory devices to override RAM devices when ROM and RAM memory are

assigned the same memory addresses. INH1/ may also be used to allow memory mapped I/O devices to override RAM memory.

INH2/

Inhibit ROM signal; prevents ROM memory devices from responding to the memory address on the system address bus. INH2/ effectively allows auxiliary ROM (e.g., a bootstrap program) to override ROM devices when ROM and auxiliary ROM memory are assigned the same memory addresses. INH2/ may also be used to allow memory mapped I/O devices to override ROM memory.

Data Lines

DAT0/ - DATF/

16 bidirectional data lines; used to transmit or receive information to or from a memory location or I/O port. DATF/ being the most significant bit. In 8-bit systems, only lines DAT0/ - DAT7/ are used (DAT7/ being the most significant bit). In 16-bit systems, either 8 or 16 lines may be used for data transmission.

Bus Priority Resolution Lines

BCLK/

Bus clock; the negative edge (high to low) of BCLK/ is used to synchronize bus priority resolution circuits. BCLK/ is asynchronous to the CPU clock. It has a 100 ns minimum period and a 35% to 65% duty cycle. BCLK/ may be slowed, stopped, or single stepped for debugging.

CCLK/

Constant clock; a bus signal which provides a clock signal of constant frequency for unspecified general use by modules on the system bus. CCLK/ has a minimum period of 100 ns and a 35% to 65% duty cycle.

BPRN/

Bus priority in signal; indicates to a particular master module that no higher priority module is requesting use of the system bus. BPRN/ is synchronized with BCLK/. This signal is not based on the backplane.

BPRO/
Bus priority out signal; used with serial (daisy chain) bus priority resolution schemes. BPRO/ is passed to the BPRN/ input of the master module with the next lower bus priority. BPRO/ is synchronized with BCLK/. This signal is not based on the backplane.

BUSY/
Bus busy signal; an open collector line driven by the bus master currently in control to indicate that the bus is currently in use. BUSY/ prevents all other master modules from gaining control of the bus. BUSY/ is synchronized with BCLK/.

BREQ/
Bus request signal; used with a parallel bus priority network to indicate that a particular master module requires use of the bus for one or more data transfers. BREQ/ is synchronized with BCLK/. This signal is not based on the backplane.

CBRQ/
Common bus request; an open-collector line which is driven by all potential bus masters and is used to inform the current bus master that another master wishes to use the bus. If CBRQ/ is high, it indicates to the bus master that no other master is requesting the bus, and therefore, the present bus master can retain the bus. This saves the bus exchange overhead for the current master.

Information Transfer Protocol Lines

A bus master provides separate read/write command signals for memory and I/O devices: MRDC/, MWTC/, IORC/ and IOWC/, as explained below. When a read/write command is active, the address signals must be stabilized at all slaves on the bus. For this reason, the protocol requires that a bus master must issue address signals (and data signals for a write operation) at least 50 ns ahead of issuing a read/write command to the bus, initiating the data transfer. The bus master must keep address signals unchanged until at least 50 ns after the read/write command is turned off, terminating the data transfer.

A bus slave must provide an acknowledge signal to

the bus master in response to a read or write command signal.

MRDC/
Memory read command; indicates that the address of a memory location has been placed on the system address lines and specifies that the contents (8 or 16 bits) of the addressed location are to be read and placed on the system data bus. MRDC/ is asynchronous with respect to BCLK/.

MWTC/
Memory write command; indicates that the address of a memory location has been placed on the system address lines and that data (8 or 16 bits) has been placed on the system data bus. MWTC/ specifies that the data is to be written into the addressed memory location. MWTC/ is asynchronous with respect to BCLK/.

IORC/
I/O read command; indicates that the address of an input port has been placed on the system address bus and that the data (8 or 16 bits) at that input port is to be read and placed on the system data bus. IORC/ is asynchronous with respect to BCLK/.

IOWC/
I/O write command; indicates that the address of an output port has been placed on the system address bus and that the contents of the system data bus (8 or 16 bits) are to be output to the address port. IOWC/ is asynchronous with respect to BCLK/.

XACK/
Transfer acknowledge signal; the required response of a slave board which indicates that the specified read/write operation has been completed. That is, data has been placed on, or accepted from, the system data bus lines. XACK/ is asynchronous with respect to BCLK/.

Asynchronous Interrupt Lines

INT0/ - INT7/
8 Multi-level, parallel interrupt request lines;

used with a parallel interrupt resolution network. INT0/ has the highest priority, while INT7/ has lowest priority. Interrupt lines should be driven with open collector drivers.

INTA/

Interrupt acknowledge; an interrupt acknowledge line (INTA/), driven by the bus master, requests the transfer of interrupt information onto the bus from slave priority interrupt controllers (8259s or 8259As). The specific information timed onto the bus depends upon the implementation of the interrupt scheme. In general, the leading edge of INTA/ indicates that the address bus is active while the trailing edge indicates that data is present on the data lines.

MULTIBUS P2 Signal Lines — The signals contained on the MULTIBUS P2 auxiliary connector are used primarily by optional power back-up circuitry for memory protection. P2 signals are not bused on the backplane, and therefore, require a separate connector for each board using the P2 signals. Present iSBC boards have a slot in the card edge and should be used with a keyed P2 edge connector. Use of the P2 signal lines is optional.

ACLO

AC Low; this signal generated by the power supply goes high when the AC line voltage drops below a certain voltage (e.g., 103v AC in 115v AC line voltage systems) indicating D.C. power will fail in 3 msec. ACLO goes low when all D.C. voltages return to approximately 95% of the regulated value. This line must be pulled up by the optional standby power source, if one is used.

PFIN/

Power fail interrupt; this signal interrupts the processor when a power failure occurs, it is driven by external power fail circuitry.

PFSN/

Power fail sense; this line is the output of a latch which indicates that a power failure has occurred. It is reset by PFSR/. The power fail

sense latch is part of external power fail circuitry and must be powered by the standby power source.

PFSR/

Power fail sense reset; this line is used to reset the power fail sense latch (PFSN/).

MPRO/

Memory protect; prevents memory operation during period of uncertain DC power, by inhibiting memory requests. MPRO/ is driven by external power fail circuitry.

ALE

Address latch enable; generated by the CPU (8085 or 8086) to provide an auxiliary address latch.

HALT/

Halt; indicates that the master CPU is halted.

AUX RESET/

Auxiliary Reset; this externally generated signal initiates a power-up sequence.

WAIT/

Bus master wait state; this signal indicates that the processor is in a wait state.

Reserved — Several P1 and P2 connector bus pins are unused. However, they should be regarded as reserved for dedicated use in future Intel products.

Power Supplies — The power supply bus pins are detailed in Appendix A which contains the pin assignment of signals on the MULTIBUS backplane.

It is the designer's responsibility to provide adequate bulk decoupling on the board to avoid current surges on the power supply lines. It is also recommended that you provide high frequency

decoupling for the logic on your board. Values of 22uF for +5v and +12v pins and 10uF for -5v and -12v pins are typical on iSBC boards.

Operating Characteristics

Beyond the definition of the MULTIBUS signals themselves, it is important to examine the operating characteristics of the bus. The AC requirements outline the timing of the bus signals and in particular, define the relationships between the various bus signals. On the other hand, the DC requirements specify the bus driver characteristics, maximum bus loading per board, and the pull-up/down resistors.

The AC requirements are best presented by a discussion of the relevant timing diagrams. Appendix B contains a list of the MULTIBUS timing specifications. The following sections will discuss data transfers, inhibit operations, interrupt operations, MULTIBUS multi-master operation and power fail considerations.

Data Transfers — Data transfers on the MULTIBUS system bus occur with a maximum bandwidth of 5 MHz for single or multiple read/write transfers. Due to bus arbitration and memory access time, a typical maximum transfer rate is often on the order of 2 MHz.

Read Data

Figure 1 shows the read operation AC timing diagram. The address must be stable (t_{AS}) for a minimum of 50 ns before command (IORC/ or MRDC/). This time is typically used by the bus interface to decode the address and thus provide the required device selects. The device selects establish the data paths on the user system in anticipation of the strobe signal (command) which will follow. The minimum command pulse width is 100 ns. The address must remain stable for at least 50 ns following the command (t_{AH}). Valid data should not be driven onto the bus prior to command, and must not be removed until the command is cleared. The XACK/ signal, which is a response indicating the specified read/write operation has been completed, must coincide or follow both the read access and valid data (t_{DXL}). XACK/ must be held until the command is cleared (t_{XAH}).

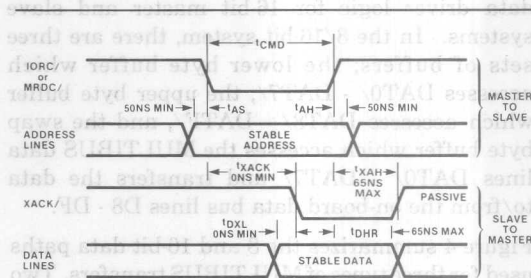


Figure 1. Read AC Timing

Write Data

The write operation AC timing diagram is shown in Figure 2. During a write data transfer, valid data must be presented simultaneously with a stable address. Thus, the write data setup time (t_{DS}) has the same requirement as the address setup time (t_{AS}). The requirement for stable data both before and after command (IOWC/ or MWTC/) enables the bus interface circuitry to latch data on either the leading or trailing edge of command.

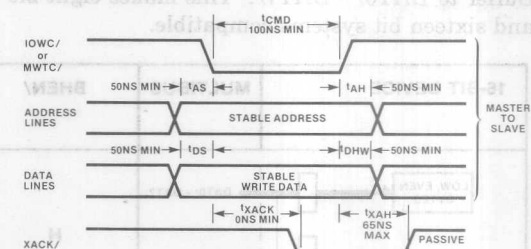


Figure 2. Write AC Timing

Data Byte Swapping in 16-bit Systems

A 16-bit master may transfer data on the MULTIBUS data lines using 8-bit or 16-bit paths depending on whether a byte or word (2 byte) operation has been specified. (A word transfer specified with an odd I/O or memory address will actually be executed as two single byte transfers.) An 8-bit master may only perform byte transfers on the MULTIBUS data lines DAT0/- DAT7/.

In order to maintain compatibility with older 8-bit masters and slaves, a byte swapping buffer is included in all new 16-bit masters and 16-bit slaves. In the iSBC product line, all byte transfers will take place on the low 8 data lines DAT0/- DAT7/. Figure 3 contains an example of 8/16-bit

data driver logic for 16-bit master and slave systems. In the 8/16-bit system, there are three sets of buffers; the lower byte buffer which accesses DAT0/- DAT7/, the upper byte buffer which accesses DAT8/- DATF/, and the swap byte buffer which accesses the MULTIBUS data lines DAT0/- DAT7/ and transfers the data to/from the on-board data bus lines D8 - DF.

Figure 4 summarizes the 8 and 16-bit data paths used for three types of MULTIBUS transfers. Two signals control the data transfers,

Byte High Enable (BHEN/) active indicates that the bus is operating in sixteen bit mode, and Address Bit 0 (ADR0/) defines an even or odd byte transfer address.

On the first type of transfer, BHEN/ is inactive, and ADR0/ is inactive indicating the transfer of an even eight bit byte. The transfer takes place across data lines DAT0/- DAT7/.

On the second type of transfer, BHEN/ is inactive, and ADR0/ is active indicating the transfer of a high (odd) byte. On this type of transfer, the odd (high) byte is transferred through the Swap Byte Buffer to DAT0/- DAT7/. This makes eight bit and sixteen bit systems compatible.

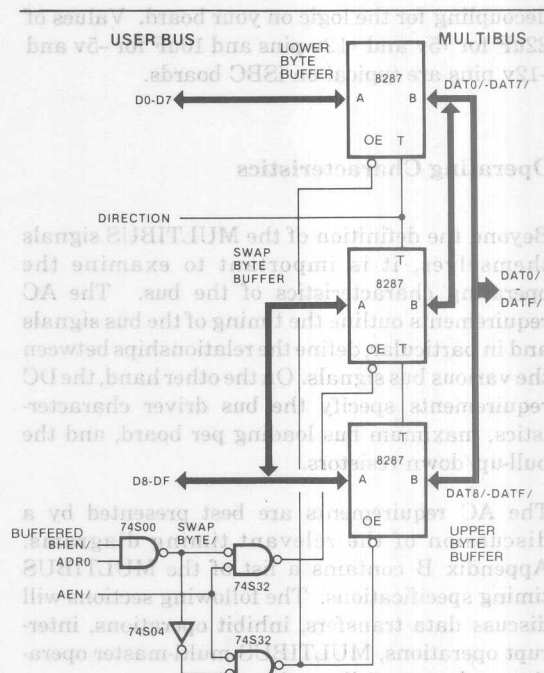


Figure 3. 8/16-Bit Data Drivers

16-BIT DEVICE	MULTIBUS	BHEN/	ADR0/	MULTIBUS TRANSFER DATA PATH	DEVICE BYTE TRANSFERRED
		H	H	8-BIT, DAT0/- DAT7/	EVEN
		H	L	8-BIT, DAT0/- DAT7/	ODD
		L	H	16-BIT, DAT0/- DATF/	EVEN AND ODD

Figure 4. 8/16-Bit Device Transfer Operation

The third type of transfer is a 16 bit (word) transfer. This is indicated by BHEN/ being active, and ADR0/ being inactive. On this type of transfer, the low (even) byte is transferred on DAT0/ - DAT7/ and the high (odd) byte is transferred on DAT8/ - DATF/.

Note that the condition when both BHEN/ and ADR0/ are active is not used with present iSBC boards. This condition could be used to transfer a high odd byte of data on DAT8/ - DATF/, thus eliminating the need for the swap byte buffer. However, this is not a recommended transfer type, because it eliminates the capability of communicating with 8-bit modules.

Inhibit Operations — Bus inhibit operations are required by certain bootstrap and memory mapped I/O configurations. The purpose of the inhibit operation is to allow a combination of RAM, ROM, or memory mapped I/O to occupy the same memory address space. In the case of a bootstrap, it may be desirable to have both ROM and RAM memory occupy the same address space, selecting ROM instead of RAM for low order memory only when the system is reset. A system designed to use

memory mapped I/O, which has actual memory occupying the memory mapped I/O address space, may need to inhibit RAM or ROM memory to perform its functions.

There are two essential requirements for a successful inhibit operation. The first is that the inhibit signal must be asserted as soon as possible, within a maximum of 100 ns (t_{CI}), after stable address. The second requirement for a successful inhibit operation is that the acknowledge must be delayed (t_{XACKB}) to allow the inhibited slave to terminate any irreversible timing operations initiated by detection of a valid command prior to its inhibit.

This situation may arise because a command can be asserted within 50 ns after stable address (t_{AS}) and yet inhibit is not required until 100 ns (t_{ID}) after stable address. The acknowledge delay time (t_{XACKB}) is a function of the cycle time of the inhibited slave memory. Inhibiting the iSBC 016 RAM board, for example, requires a minimum of 1.5 μ sec. Less time is typically needed to inhibit other memory modules. For example, the iSBC 104 board requires 475 ns.

Figure 5 depicts a situation in which both RAM

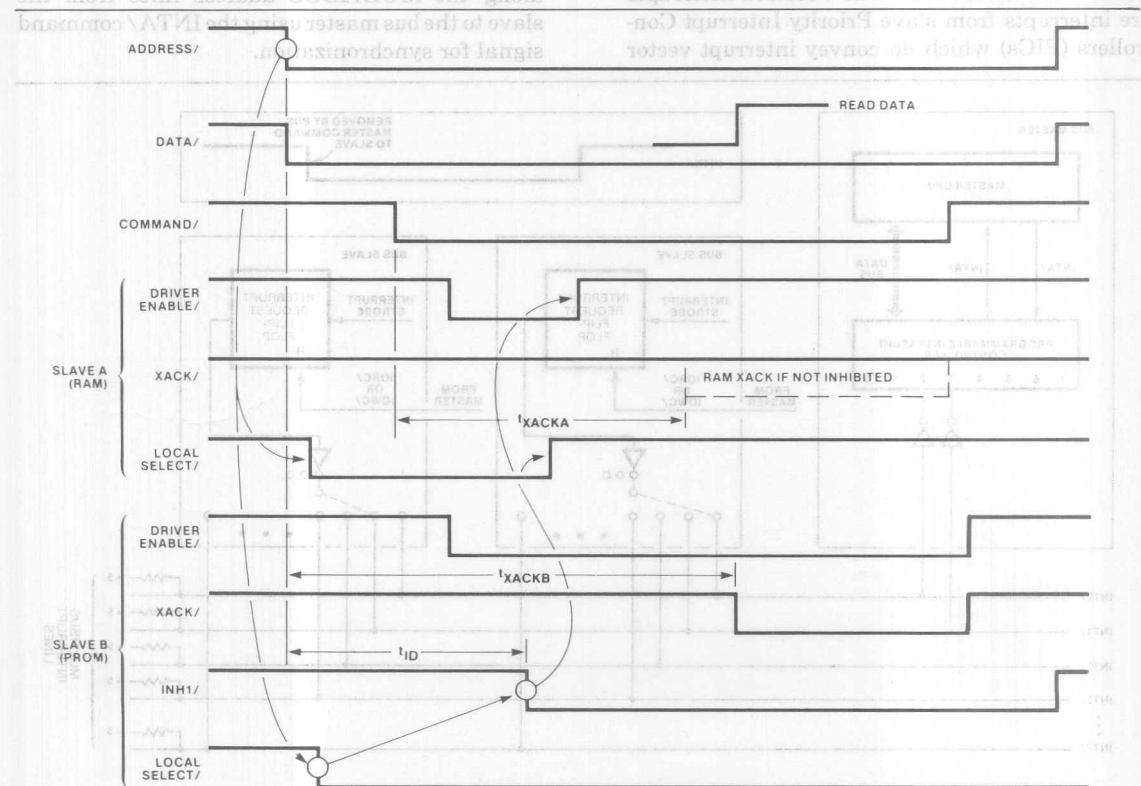


Figure 5. Inhibit Timing

(tXACKA).

trollers (PICs) which do convey interrupt vector

address information on the bus.

Non Bus Vectored Interrupts

Vectored Interrupt implementation.

Bus Vectored Interrupts

signal for synchronization.

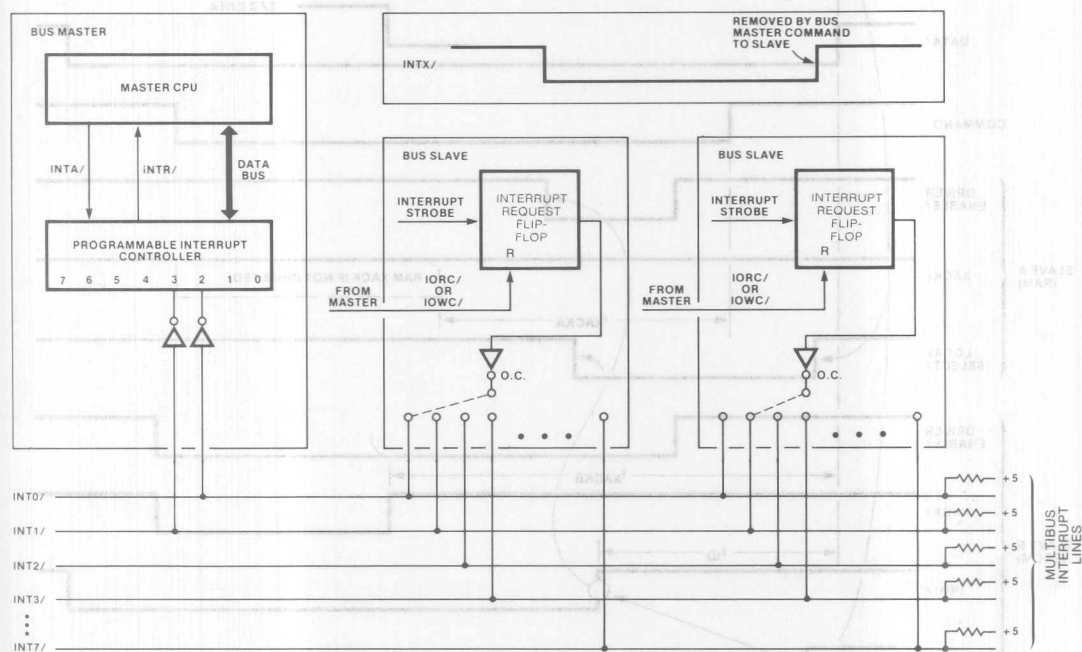


Figure 6. Non Bus Vectored Interrupt Implementation

the bus master to service the interrupt.

The MULTIBUS specification provides for only one type of Bus Vectored Interrupt operation in a given system. Slave boards which have an 8259 interrupt controller are only capable of 3 INTA/ operation (2 additional INTA/s after the first INTA/). Slave boards with the 8259A interrupt controller are capable of either 2 INTA/ or 3 INTA/ operation. All slave boards in a given system must operate in the same way (2 INTA/s or 3 INTA/s) if Bus Vectored Interrupts are to be used. However, the MULTIBUS specification does provide for Bus Vectored Interrupts and Non Bus Vectored Interrupts in the same system.

MULTIBUS Multi-Master Operation — The MULTIBUS system bus can accommodate several bus masters on the same system, each one taking control of the bus as it needs to affect data transfers. The bus masters request bus control through a bus exchange sequence.

Two bus exchange priority resolution techniques are discussed, a serial technique and a parallel technique. Figures 8 and 9 illustrate these two techniques. The bus exchange operation discussed later is the same for both techniques.

Serial Priority Technique

Serial priority resolution is accomplished with a daisy chain technique (see Figure 8). The priority input (BPRN/) of the highest priority master is tied to ground. The priority output (BPRO/) of the

request will set its BPRO/ signal high to the next lower priority master. Any master seeing a high signal on its BPRN/ line will set its BPRO/ line high, thus passing down priority information to lower priority masters. In this implementation, the bus request line (BREQ/) is not used outside of the individual masters. A limited number of masters can be accommodated by this technique, due to gate delays through the daisy chain. Using the current Intel MULTIBUS controller chip on the master boards up to 3 masters may be accommodated if a BCLK/ period of 100 ns is used. If more bus masters are required, either BCLK/ must be slowed or a parallel priority technique used.

Parallel Priority Technique

In the parallel priority technique, the priority is resolved in a priority resolution circuit in which the highest priority BREQ/ input is encoded with a priority encoder chip (74148). This coded value is then decoded with a priority decoder chip (74S138) to activate the appropriate BPRN/ line. The BPRO/ lines are not used in the parallel priority scheme. However, since the MULTIBUS backplane contains a trace from the BPRN/ signal of one card slot to the BPRO/ signal of the adjacent lower card slot, the BPRO/ must be disconnected from the bus on the board or the backplane trace must be cut. A practical limit of sixteen masters can be accommodated using the parallel priority technique due to physical bus length limitations. Figure 9 contains the schematic for a typical parallel resolution network. Note that the parallel priority resolution network must be externally supplied.

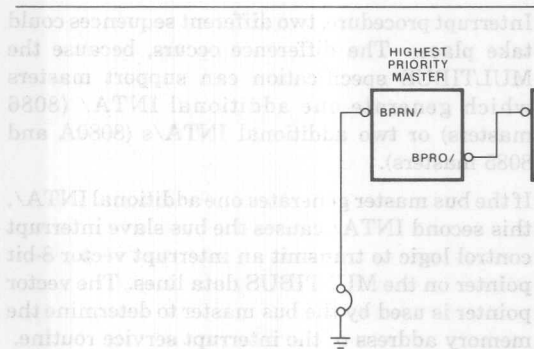


Figure 8. Serial Priority Technique

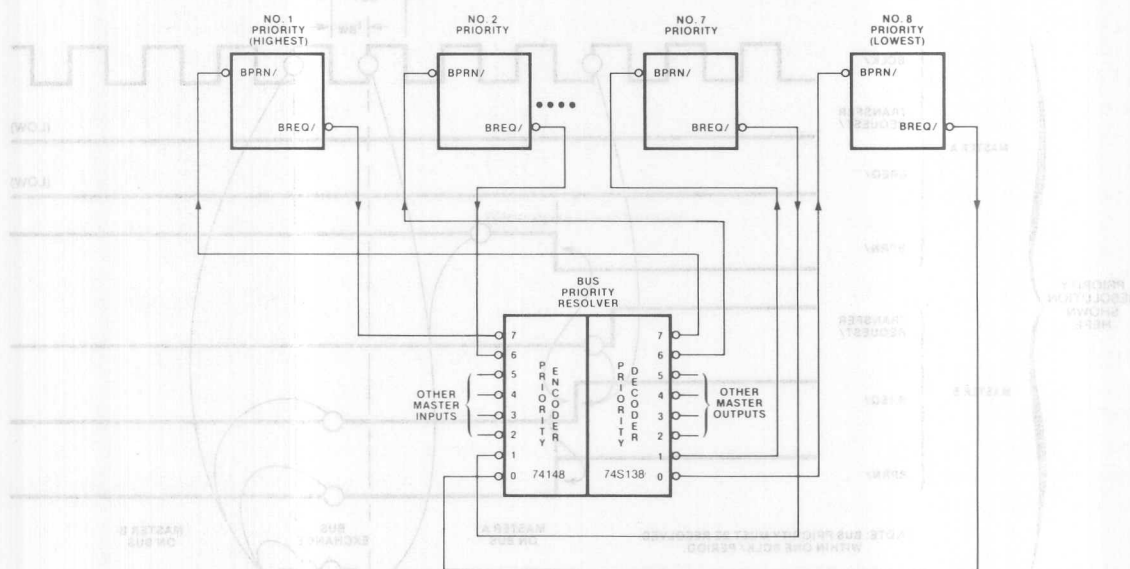


Figure 9. Parallel Priority Technique

MULTIBUS Exchange Operation — A timing diagram for the MULTIBUS exchange operation is shown in Figure 10. This implementation example uses a parallel resolution scheme, however, the timing would be basically the same for the serial resolution scheme.

In this example, master A has been assigned a lower priority than master B. The bus exchange occurs because master B generates a bus request during a time when master A has control of the bus.

The exchange process begins when master B requires the bus to access some resource such as an I/O or memory module while master A controls the bus. This internal request is synchronized with the trailing edge (high to low) of BCLK/ to generate a bus request (BREQ/). The bus priority resolution circuit changes the BPRN/ signal from active (low) to inactive (high) for master A and from inactive to active for master B. Master A must first complete the current bus command if one is in operation. After master A completes the command, it sets BUSY/ inactive on the next trailing edge of BCLK/. This allows the actual bus exchange to occur, because master A has relinquished control of the bus, and master B has been granted its BPRN/. During this time, the drivers

for master A are disabled. Master B must take control of the bus with the next trailing edge of BCLK/ to complete the bus exchange. Master B takes control by activating BUSY/ and enabling its drivers.

It is possible for master A to retain control of the bus and prevent master B from getting control. Master A activates the Bus Override (or Bus Lock) signal which keeps BUSY/ active allowing control of the bus to stay with master A. This guarantees a master consecutive bus cycles for software or hardware functions which require exclusive, continuous access to the bus.

Note that in systems with only a single master it is necessary to ground the BPRN/ pin of the master, if slave boards are to be accessed. In single board systems which use a CPU board capable of Bus Vectored Interrupt operation, the BPRN/ pin must also be grounded.

In a single master system bus transfer efficiency may be gained if the BUS OVERRIDE signal is kept active continuously. This permits the master to maintain control of the bus at all times, therefore saving the overhead of the master reacquiring the bus each time it is needed.

The CBRQ/ line may be used by a master in control of the bus to determine if another master

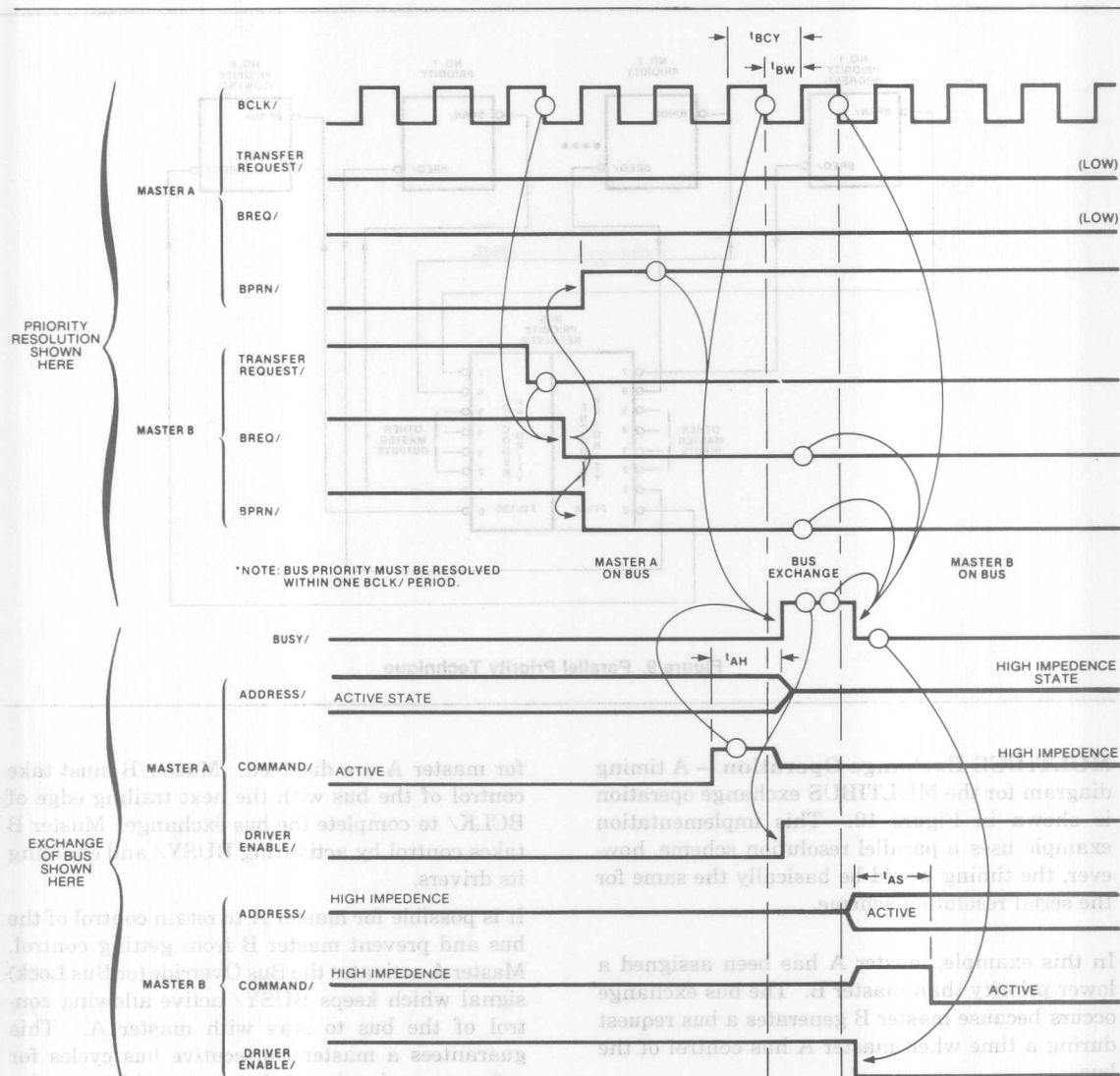


Figure 10. Bus Control Exchange Operation

requires the bus. If a master currently in control of the bus sees the CBRQ/ line inactive, it will maintain control of the bus between adjacent bus accesses. Therefore, when a bus access is required, the master saves the overhead of reacquiring the bus. If a current bus master sees the CBRQ/ line active, it will then relinquish control of the bus after the current bus access and will contend for the bus with the other master(s) requiring the bus. The relative priorities of the masters will determine which master receives the bus.

Note that except for the BUS OVERRIDE state, no single master may keep exclusive control of the bus. This is true because it is impossible for the CPU on a master to require continuous access to the bus. Other lower priority masters will always be able to gain access to the bus between accesses of a higher priority master.

Power Fail Considerations—The MULTIBUS P2 connector signals provide a means of handling power failures. The circuits required for power

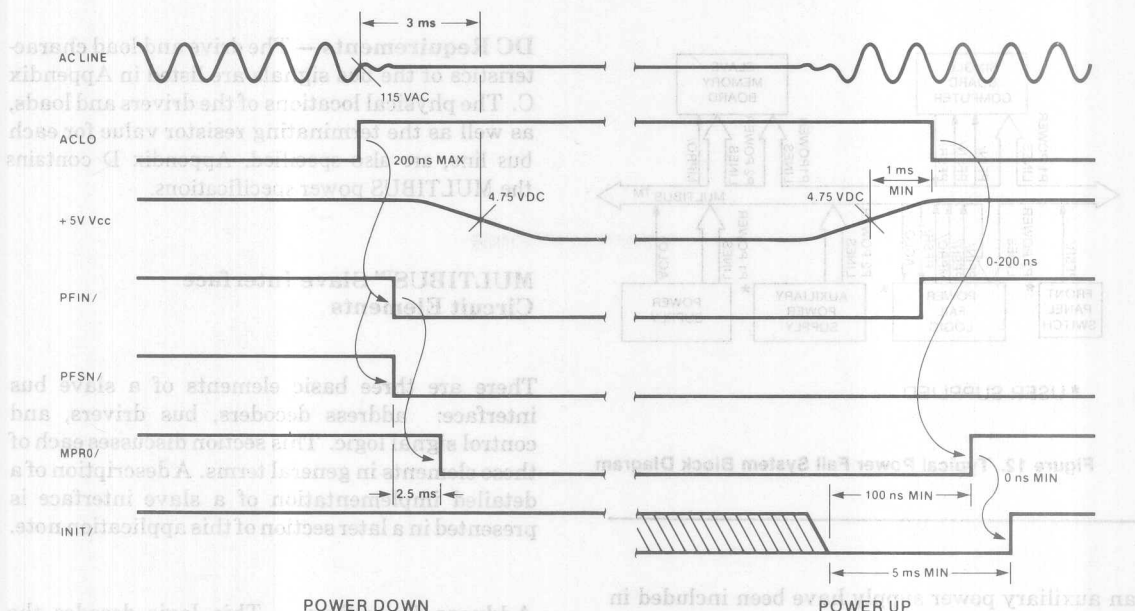


Figure 11. Power Fail Timing Sequence

failure detection and handling are optional and must be supplied by the user. Figure 11 shows the timing of a power fail sequence.

The power supply monitors the AC power level. When power drops below an acceptable value, the power supply raises ACLO which tells the power fail logic that a minimum of three milliseconds will elapse before DC power will fall below regulated voltage levels. The power fail logic sets a sense latch (PFSN/) and generates an interrupt (PFIN/) to the processor so the processor can store its environment. After a 2.5 millisecond timeout, the memory protect signal (MPRO/) is asserted by the power fail logic preventing any memory activity. As power falls, the memory goes on standby power. Note that the power fail logic must be powered from the standby source.

As the AC line revives, the logic voltage level is monitored by the power supply. After power has been at its operating level for one millisecond minimum, the power supply sets the signal ACLO low, beginning the restart sequence. First, the memory protect line (MPRO/) then the initialize line (INIT/) become inactive. The bus master now starts running. The bus master checks the power fail latch (PFSN/) and, if it finds it set, branches to

a power up routine which resets the latch (PFSR/), restores the environment, and resumes execution.

Note that INIT/ is activated only after DC power has risen to the regulated voltage levels and must stay low for five milliseconds minimum before the system is allowed to restart. Alternatively, INIT/ may be held low through an open collector device by MPRO/.

How the power failure equipment is configured is left to the system designer. The backup power source may be batteries located on the memory boards or more elaborate facilities located off-board. The location of the power fail logic determines which MULTIBUS power fail lines are used. Pins on the P2 connector have been specified for the power failure functions for use as needed.

To further clarify the location and use of the power fail circuitry, an example of a typical power fail system block diagram is shown in Figure 12. A single board computer and a slave memory board are contained in the system. It is desired to power the memory circuit elements of the memory board from auxiliary power. The single board computer will remain on the main power supply. To accomplish this, user supplied power fail logic and

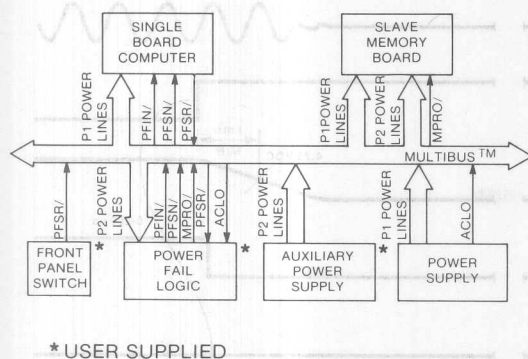


Figure 12. Typical Power Fail System Block Diagram

an auxiliary power supply have been included in the system.

The single board computer is powered from the P1 power lines and accesses the P2 signal lines PFIN/, PFSN/ and PFSR/ (only the P2 signal lines used by a particular functional block are shown on the block diagram). The PFSR/ line is driven from two sources: a front panel switch and the single board computer. The front panel switch is used during normal power-up to reset the power fail sense latch. The single board computer uses the PFSR/ line to reset the latch during a power-up sequence after a power failure. Current single board computers must access the PFSN/ and PFSR/ signals either directly with dedicated circuitry and a P2 pin connection or through the parallel I/O lines with a cable connection from the parallel I/O connector to the P2 connector.

The slave memory board uses both the P1 and P2 power lines, the P2 power lines are used (at all times) to power the memory circuit elements and other support circuits, the P1 power lines power all other circuitry. In addition, the MPRO/ line is input and used to sense when memory contents should be protected.

The power fail logic contains the power fail sense latch, and uses the PFSR/ and ACLO lines for inputs and the PFIN/ PFSN/, and MPRO/ lines for outputs. The power fail logic must be powered by the P2 power lines.

DC Requirements — The drive and load characteristics of the bus signals are listed in Appendix C. The physical locations of the drivers and loads, as well as the terminating resistor value for each bus line, are also specified. Appendix D contains the MULTIBUS power specifications.

MULTIBUS™ Slave Interface Circuit Elements

There are three basic elements of a slave bus interface: address decoders, bus drivers, and control signal logic. This section discusses each of these elements in general terms. A description of a detailed implementation of a slave interface is presented in a later section of this application note.

Address Decoding — This logic decodes the appropriate MULTIBUS address bits into RAM requests, ROM requests, or I/O selects. Care must be taken in the design of the address decode logic to ensure flexibility in the selection of base address assignments. Without this flexibility, restrictions may be placed upon various system configurations. Ideally, switches and jumper connections should be associated with the decode logic to permit field modification of base address assignments.

The initial step in designing the address decode portion of a MULTIBUS interface is to determine the required number of unique address locations. This decision is influenced by the fact that address decoding is usually done in two stages. The first stage decodes the base address, producing an enable for the second stage which generates the actual device selects for the user logic. A convenient implementation of this two stage decoding scheme utilizes a pair of decoders driven by the high order bits of the address for the first stage and a second decoder for the low order bits of the address bus. This technique forces the number of unique address locations to be a power of two, based at the address decoded by the first stage. Consider the scheme illustrated in Figure 13.

As shown in Figure 13, the address bits A₄-A₃ are used to produce switch selected outputs of the first stage of decoding. The 1 out of 8 binary decoders

have been used. The top decoder decodes address lines A₄ - A₇, and the bottom decoder decodes address lines A₈ - A_B. If only address lines A₀ - A₇ are being used for device selection, as in the case of I/O port selection in 8-bit systems, the bottom decoder may be disabled by setting switch S₂ to the ground position. Address lines A₇ and A_B drive enable inputs E₂ or E₃ of the decoders. The address lines A₀ - A₃ enter the second stage address decoder to produce 8 user device selects. The second stage decoder must first be enabled by an address that corresponds to the switch-selected base address.

Address decoding must be completed before the arrival of a command. Since the command may become active within 50 ns after stable address, the decode logic should be kept simple with a minimal number of layers of logic. Furthermore, the timing is extremely critical in systems which make use of the inhibit lines.

A linear or unary select scheme in which no binary encoding of device address (e.g., address bit A₀ selects device 0, address bit A₁ selects device 1, etc.) is performed is not recommended because the scheme offers no protection in case multiple

devices are simultaneously selected, and because the addressing within such a system is restricted by the extent of the address space occupied by such a scheme.

Data Bus Drivers — For user designed logic which simply receives data from the MULTIBUS data lines, this portion of the bus interface logic may only consist of buffers. Buffers are required to ensure that maximum allowable bus loading is not exceeded by the user logic.

In systems where the user designed logic must place data onto the MULTIBUS data lines, three-state drivers are required. These drivers should be enabled only when a memory read command (MRDC/) or an I/O read command (IORC/) is present and the module has been addressed.

When both the read and write functions are required, parallel bidirectional bus drivers (e.g., Intel 8226, 8287, etc.) are used. A note of caution must be included for the designer who uses this type of device. A problem may arise if data hold time requirements must be satisfied for user logic following write operations. When bus commands are used to directly produce both the chip select for the bidirectional bus driver and a strobe to a latch in the user logic, removal of that signal may not provide the user's latch with adequate data hold time. Depending on the specifics of the user logic, this problem may be solved by permanently enabling the data buffer's receiver circuits and controlling only the direction of the buffers.

Control Signal Logic — The control signal logic consists of the circuits that forward the I/O and memory read/write commands to their respective destinations, provide the bus with a transfer acknowledge response, and drive the system interrupt lines.

Bus Command Lines

The MULTIBUS information transfer protocol lines (MRDC/, MWTC/, IORD/, and IOWC/) should be buffered by devices with very high speed switching. Because the bus DC requirements specify that each board may load these lines with 2.0 mA, Schottky devices are recommended. LS devices are not recommended due to their poor noise immunity. The commands should be gated

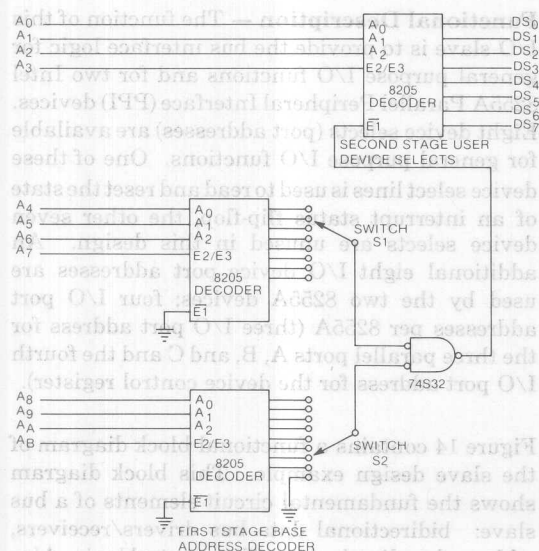


Figure 13. Two Stage Decoding Scheme

with a signal indicating the base address has been decoded to generate read and write strobes for the user logic.

Transfer Acknowledge Generation

The user interface transfer acknowledge generation logic provides a transfer acknowledge response, XACK/, to notify the bus master that write data provided by the bus master has been accepted or that read data it has requested is available on the MULTIBUS data lines. XACK/ allows the bus master to conclude its current instruction.

Since XACK/ timing requirements depend on both the CPU of the bus master and characteristics of the user logic, a circuit is needed which will provide a range of easily modified acknowledge responses. The transfer acknowledge signals must be driven by three-state drivers which are enabled when the bus interface is addressed and a command is present.

Interrupt Signal Lines

The asynchronous interrupt lines must be driven by open collector devices with a minimum drive of 16 mA.

In a typical Non Bus Vectored Interrupt system, logic must be provided to assert and latch-up an interrupt signal. In addition to driving the MULTIBUS interrupt lines, the latched interrupt signal would be read by an I/O operation such as reading the module's status. The interrupt signal would be cleared by writing to the status register.

III. MULTIBUS™ SLAVE DESIGN EXAMPLE

A MULTIBUS slave design example has been included in this application note to reinforce the theory previously discussed. The design example is of general purpose I/O slave interface. This design example could easily be modified to be used as a slave memory interface by buffering the address signals and using the appropriate MULTIBUS memory commands. In addition, to help the reader better understand an application for an I/O slave interface, two Intel 8255A Parallel Peripheral Interface (PPI) devices are shown connected to the slave interface.

The design example is shown in both 8/16-bit version and an 8-bit version. The 8/16-bit version

is an I/O interface which will permit a 16-bit master to perform 8 or 16 bit data transfers. 8-bit masters may also use the 8/16-bit version of the design example to perform 8-bit data transfers.

The 8-bit version of the design example may be used by both 8 or 16-bit masters, but will only perform 8-bit data transfers. It does not contain the circuitry required to perform 16-bit data transfers.

Both the 8/16-bit version and the 8-bit version of the design example were implemented on an iSBC 905 prototype board. The schematics for each of the examples are given in Appendices F and G.

Functional/Programming Characteristics

This section describes the organization of the slave interface from two points of view, the functional point of view and the programming characteristics. First, the principal functions performed by the hardware are identified and the general data flow is illustrated. This point of view is intended as an introduction to the detailed description provided in the next section; Theory of Operation. In the second point of view, the information needed by a programmer to access the slave is summarized.

Functional Description — The function of this I/O slave is to provide the bus interface logic for general purpose I/O functions and for two Intel 8255A Parallel Peripheral Interface (PPI) devices. Eight device selects (port addresses) are available for general purpose I/O functions. One of these device select lines is used to read and reset the state of an interrupt status flip-flop, the other seven device selects are unused in this design. An additional eight I/O device port addresses are used by the two 8255A devices; four I/O port addresses per 8255A (three I/O port address for the three parallel ports A, B, and C and the fourth I/O port address for the device control register).

Figure 14 contains a functional block diagram of the slave design example. This block diagram shows the fundamental circuit elements of a bus slave: bidirectional data bus drivers/receivers, address decoding logic and bus control logic. Also shown is the address decoding logic for the low order four bits, the interrupt logic which is selected by this decoding logic, and the two 8255A devices.

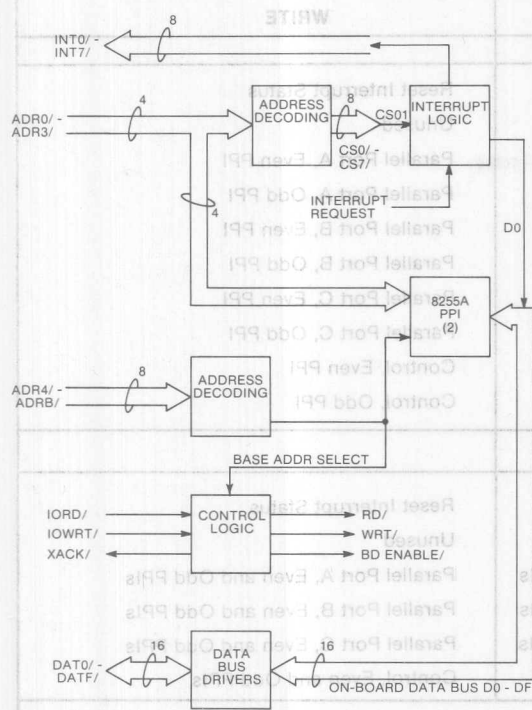


Figure 14. MULTIBUS™ Slave Design Example Functional Block Diagram

Programming Characteristics — The slave design example provides 16 I/O port addresses which may be accessed by user software. The base address of the 16 contiguous port addresses is selected by wire wrap connections on the prototype board. The wire wrap connections specify address bits ADR4/- ADRB/. They allow the selection of a base address on any 16 byte boundary. Twelve address bits (ADR0/- ADRB/) are used since 16-bit (8086 based) masters use 12 bits to specify I/O port addresses. If an 8 bit (8080 or 8085 based) master is used with this slave board, the high order address bits (ADR8/- ADRB/) must not be used by the decoding circuits; a wire wrap jumper position (ground position) is provided for this.

The 16 I/O port addresses are divided into two groups of 8 port addresses by decoding address line ADR3/. Port addresses XX0 - XX7 are used for general I/O functions (XX indicates any hexadecimal digit combination). Port address XX0 is used for accessing the interrupt status flip-flop and

port addresses XX1 - XX7 are not used in this example. Port addresses XX8 - XXF are used for accessing the PPIs. If port addresses XX8 - XXF are selected, then ADR0/ is used to specify which of two PPIs are selected. If the address is even (XX8, XXA, XXC, or XXE) then one PPI is selected. If the address is odd (XX9, XXB, XXD, or XXF), then the other PPI is selected. ADR1/ and ADR2/ are connected directly to the PPIs. Table 1 summarizes the I/O port addresses of the slave design example. Note that if a 16-bit master is used, it is possible to access the slave in a byte or word mode. If word access is used with port address XX8, XXA, XXC, or XXE, then 16 bit transfers will occur between the PPIs and the master. These 16 bit transfers occur because an even address has been specified and the MULTI-BUS BHEN/ signal indicates that a 16-bit transfer is requested.

Theory of Operation

In the preceding section, each of the slave design example functional blocks was identified and briefly explained. This section explains how these functions are implemented. For detailed circuit information, refer to the schematics in Appendices F and G. The schematic in Appendix F is on a foldout page so that the following text may easily be related to the schematic.

The discussion of the theory of operation is divided into five segments, each of which discusses a different function performed by the MULTIBUS slave design example. The five segments are:

1. Bus address decoding
2. Data buffers
3. Control signals
4. Interrupt logic
5. PPI operation

Each of these topics are discussed with regard to the 8/16-bit version of the design example; followed by a discussion of the circuit elements which are required by the 8-bit version of the interface.

Bus Address Decoding — Bus address decoding is performed by two 8205 1 out of 8 binary decoders. One decoder (A3) decodes address bits ADR8/- ADRB/ and the second decoder (A2) decodes address bits ADR4/- ADR7/. The base address

Table 1

SLAVE DESIGN EXAMPLE PORT ADDRESSES

I/O PORT ADDRESS	READ	WRITE
BYTE ACCESS		
XX0	Bit 0 = Interrupt Status	Reset Interrupt Status
XX1 - XX7	Unused	Unused
XX8	Parallel Port A, Even PPI	Parallel Port A, Even PPI
XX9	Parallel Port A, Odd PPI	Parallel Port A, Odd PPI
XXA	Parallel Port B, Even PPI	Parallel Port B, Even PPI
XXB	Parallel Port B, Odd PPI	Parallel Port B, Odd PPI
XXC	Parallel Port C, Even PPI	Parallel Port C, Even PPI
XXD	Parallel Port C, Odd PPI	Parallel Port C, Odd PPI
XXE	Illegal Condition	Control, Even PPI
XXF	Illegal Condition	Control, Odd PPI
WORD ACCESS		
XX0	Bit 0 = Interrupt Status	Reset Interrupt Status
XX2 - XX6	Unused	Unused
XX8	Parallel Port A, Even and Odd PPIs	Parallel Port A, Even and Odd PPIs
XXA	Parallel Port B, Even and Odd PPIs	Parallel Port B, Even and Odd PPIs
XXC	Parallel Port C, Even and Odd PPIs	Parallel Port C, Even and Odd PPIs
XXE	Illegal Condition	Control, Even and Odd PPIs

XX = Any hex digits, assigned by jumpers; XX defines the base address.

selected is determined by the position of wire wrap jumpers. The outputs of the two decoders are ANDed together to form the BASE ADR SELECT/ signal. This signal specifies the base address for a group of 16 I/O ports. Using the wire wrap jumper positions shown in the schematic, a base address of E3 has been selected. Therefore, this MULTIBUS slave board will respond to I/O port addresses in the E30 - E3F range.

If this slave board is to be used with 8-bit MULTIBUS masters, the high order address bits must not be decoded. Therefore, the wire wrap jumper which selects the output of decoder A3 must be placed in the top (ground) position (pin 10 of gate A9 to ground).

The low order 4 address lines (ADR0/- ADR3/) are buffered and inverted using 74LS04 inverters. These address lines are input to an 8205 for decoding a chip select for the interrupt logic; the address lines are also used directly by the PPIs. LS-Series logic is required for buffering to meet the MULTIBUS specification for I_{IL} (low level input

current). S-Series or standard series logic will not meet this specification.

Address decoder A4 is used to decode addresses E30 - E37. The CS0/ output of this decoder is used to select the interrupt logic, thus I/O port address E30 is used to read and reset the interrupt latch. The remaining outputs from decoder A4 (CS1/- CS7/) are not used in this example. They would normally be used to select other functions in a slave board with more capability. Note that in the schematic shown in Appendix G for the 8-bit version of this slave design example, the high order (ADR8/- ADRB/) address decoder is not included and the BHEN/ signal is not used.

Data Buffers — Intel 8287 8-bit parallel bi-directional bus drivers are used for the MULTIBUS data lines DAT0/- DATF/. In the 8/16-bit version of the slave board, three 8287 drivers are used.

When an 8-bit data transfer is requested, either driver A5, which is connected to on-board data

on-board data lines D8 - DF, is used. If a byte transfer is requested from an even address, driver A5 will be selected. If a byte transfer from an odd address is requested, driver A6 will be selected. All byte transfers take place on MULTIBUS data lines DAT0/ - DAT7/. When a word (16-bit) transfer is requested from an even address, drivers A5 and A7 will be used. Note that if a user program requests a word transfer from an odd address, 16-bit masters in the iSBC product line will actually perform two byte transfer requests.

The logic which determines the chip selection (8287 input signal OE, output enable) signals for the bus drivers uses the low order address bit (ADR0/) and the buffered Byte High Enable signal (BHENBL/). Note that the MULTIBUS signal BHEN/ has been buffered with an 74LS04 inverter. This is done to meet the bus address line loading specification. The SWAP BYTE/ signal which is generated is qualified by the BD ENBL/ signal and used to select the bus drivers.

The steering pin for the 8287 drivers is labelled T (transmit) and is driven by the signal RD. When an input (read) request is active or when neither a read or write command is being serviced, the direction of data transfer of the 8287 will be set for B to A.

The 8287 drivers are set to point IN (direction B to A) when no MULTIBUS I/O transfer command is being serviced for two reasons. First, if the driver were pointed OUT (direction A to B) and a write command occurred, it would be necessary to turn the buffers IN and set the OE (output enable) signal active before the data could be transferred to the on-board bus. A possibility of a "buffer-fight" could occur in some designs if the OE signal permitted an 8287 to drive the MULTIBUS data lines momentarily before the steering signal could switch the direction of the 8287. In this case, both the MULTIBUS master and the slave would be driving the data lines; this is not recommended. (In this particular design, the steering signal will always stabilize before the OE signal becomes active.)

The second reason the driver is pointing IN when no command is present is due to the "data valid after WRITE" requirements of the 8255As. The 8255A requires that data remain on its data lines for 30 ns after the WRITE command (WR at the 8255A) is removed. This requirement will be met if the direction of the 8287 drivers is not switched

(WR1/ could have been used to steer the 8287 instead of RD); and if the capacitance of the on-board data bus lines is sufficient to hold the data values on the bus after the 8287 OE signal and the 8255A PPI WRT/ signal go inactive. The on-board data bus may easily be designed such that the capacitance of the lines is sufficient to meet the 30 ns data hold time requirement. In addition, the current leakage of all devices connected to the on-board bus must be kept small to meet the 30 ns data hold time requirement.

The 8-bit version of this design example uses only one 8287 instead of the three required by the 8/16-bit version. The logic required to control the swap byte buffer is also not necessary. The chip select signal used for the 8287 is the BD ENBL/ signal.

Control Signals — The MULTIBUS control signals used by this slave design example are IORC/, IOWC/, and XACK/. IORC/ and IOWC/ are qualified by the BASE ADR SELECT/ signal to form the signals RD and WRT. RD and WRT are used to drive the interrupt logic, the PPI logic and the XACK/ (transfer acknowledge) logic.

For the XACK/ logic RD and WRT are ORed to form the BD ENBL/ signal which is inverted and used to drive the CLEAR pin of a shift register. When the slave board is not being accessed, the CLEAR pin of the shift register will be low (BD ENBL/ is high). This causes the shift register to remain cleared and all outputs of the shift register will be low. When the slave board is accessed, the CLEAR pin will be high, and the A and B inputs (which are high) will be clocked to the output pins by CCLK/. To select a delay for the XACK/ signal, a jumper must be installed from one of the shift register output pins to the 8089 tri-state driver. Each of the shift register output pins select an integer multiple of CCLK/ periods for the signal delay. Since the CCLK/ signal is asynchronous, the actual delay selected may only be specified with a tolerance of one CCLK/ period. In this example a delay of 3 - 4 CCLK/ periods was selected; with a CCLK/ period of 100 ns, the XACK/ delay would occur somewhere within the range of 300 - 400 ns from the time when the CLEAR signal goes high.

The control signal logic used in the 8-bit version of the slave design example is identical to the logic used in the 8/16-bit version.

Interrupt Logic — The interrupt logic uses a 74S74 flip-flop to latch an asynchronous interrupt request from some external logic. The Q output of the INTERRUPT REQUEST LATCH is output through an open collector gate to one of the MULTIBUS interrupt lines. The state of the INTERRUPT REQUEST LATCH is transferred to the INTERRUPT STATUS LATCH when a read command is performed on I/O port BASE ADDRESS+0 (E30 for the jumper configuration shown). The Q output of INTERRUPT STATUS LATCH is used to drive data line D0 of the on-board data bus by using an 8089 tri-state driver. If a user program performs an INPUT from I/O port E30, data bit 0 will be set to 1 if the INTERRUPT REQUEST LATCH is set.

The purpose of INTERRUPT STATUS LATCH is to minimize the possibility of the asynchronous interrupt occurring while the interrupt status is being read by a bus master. If the latch was not included in the design and an asynchronous interrupt did occur while a bus master is reading MULTIBUS data line DAT0/, a data buffer on the master could go into a meta-stable state. By adding the extra latch, which is clocked by the IORD/ command for I/O port E30, the possibility of data line DAT0/ changing during a bus master read operation is eliminated. The INTERRUPT REQUEST LATCH is cleared when a user program performs an OUTPUT to I/O port E30.

This interrupt structure assumes that several interrupt sources may exist on the same MULTIBUS interrupt line (for example, INT3/). When the MULTIBUS master gets interrupted, it must poll the possible sources of the interrupt received and after determining the source of the interrupt, it must clear the INTERRUPT REQUEST LATCH for that particular interrupt source.

The interrupt logic for the 8-bit version of the design example is identical to the interrupt logic of the 8/16-bit version of the design example.

PPI Operation — Two 8255A Parallel Peripheral Interface (PPI) devices are shown interfaced to the slave design example logic. One PPI is connected to the on-board data bus lines D0 - D7 and is addressed with the even I/O port addresses E38, E3A, E3C, and E3E. The second PPI is connected to data bus lines D8 - DF and is addressed with the odd I/O port addresses E39, E3B,

E3D, and E3F. The even or odd I/O port selection is controlled by using the ADR0 address line in the chip select term of the PPIs. In addition, the odd PPI (A11) is selected when the BHENBL term is high. This occurs when the MULTIBUS signal BHEN/ is low indicating that a word (16-bit) I/O instruction is being executed. When a word I/O instruction is executed, both PPIs will perform the I/O operation specified.

The specifications of the 8255A device state that the address lines A0 and A1 and the chip select lines must be stable before the \overline{RD} or \overline{WR} lines are activated. The MULTIBUS specification address set-up time of 50 ns and the short gate propagation delays in this design assure that the address lines are stable before \overline{RD} or \overline{WR} are active.

The data hold requirements of the 8255A were discussed in a previous section. The 8255A specification states that data will be stable on the data bus lines a maximum of 250 ns after a READ command. This specification was used to select the delay for the XACK/ signal.

The PPI operation for the 8-bit version of the design example is slightly different than that used for the 8/16-bit version. The chip select signal for the bottom PPI does not use the BHENBL term since 16-bit data transfers are not possible with an 8-bit I/O slave board. Also, the chip select and address signals have been swapped so the top PPI occupies I/O address range X8 - XB, and the bottom PPI occupies I/O address range XC - XF (X is the base address of the 8-bit version). This swapping of the address lines was not necessary; however, it was thought to be more convenient to access the PPIs in two groups of 4 contiguous I/O port addresses.

IV. SUMMARY

This application note has shown the structure of the Intel MULTIBUS system bus. The structure supports a wide range of system modules from the Intel OEM Microcomputer Systems product line that can be extended with the addition of user designed modules. Because the user designed modules are no doubt unique to particular applications, a goal of this application note has been to describe in detail the singular common element - the bus interface. Material has also been presented to assist the systems designer to understanding the bus functions so that successful systems integration can be achieved.

APPENDIX A

PIN ASSIGNMENT OF BUS SIGNALS ON MULTIBUS BOARD P1 CONNECTOR

	(COMPONENT SIDE)			(CIRCUIT SIDE)		
	PIN	MNEMONIC	DESCRIPTION	PIN	MNEMONIC	DESCRIPTION
POWER SUPPLIES	1	GND	Signal GND	2	GND	Signal GND
	3	+5V	+5Vdc	4	+5V	+5Vdc
	5	+5V	+5Vdc	6	+5V	+5Vdc
	7	+12V	+12Vdc	8	+12V	+12Vdc
	9	-5V	-5Vdc	10	-5V	-5Vdc
	11	GND	Signal GND	12	GND	Signal GND
BUS CONTROLS	13	BCLK/	Bus Clock	14	INIT/	Initialize
	15	BPRN/	Bus Pri. In	16	BPRO/	Bus Pri. Out
	17	BUSY/	Bus Busy	18	BREQ/	Bus Request
	19	MRDC/	Mem Read Cmd	20	MWTC/	Mem Write Cmd
	21	IORC/	I/O Read Cmd	22	IOWC/	I/O Write Cmd
	23	XACK/	XFER Acknowledge	24	INH1/	Inhibit 1 disable RAM
BUS CONTROLS AND ADDRESS	25		Reserved	26	INH2/	Inhibit 2 disable PROM or ROM
	27	BHEN/	Byte High Enable	28	AD10/	Address Bus
	29	CBRQ/	Common Bus Request	30	AD11/	
	31	CCLK/	Constant Clk	32	AD12/	
	33	INTA/	Intr Acknowledge	34	AD13/	
INTERRUPTS	35	INT6/	Parallel Interrupt Requests	36	INT7/	Parallel Interrupt Requests
	37	INT4/		38	INT5/	
	39	INT2/		40	INT3/	
	41	INT0/		42	INT1/	
ADDRESS	43	ADRE/	Address Bus	44	ADRF/	Address Bus
	45	ADRC/		46	ADRD/	
	47	ADRA/		48	ADRB/	
	49	ADR8/		50	ADR9/	
	51	ADR6/		52	ADR7/	
	53	ADR4/		54	ADR5/	
	55	ADR2/		56	ADR3/	
	57	ADR0/		58	ADR1/	
DATA	59	DATE/	Data Bus	60	DATF/	Data Bus
	61	DATC/		62	DATD/	
	63	DATA/		64	DATB/	
	65	DAT8/		66	DAT9/	
	67	DAT6/		68	DAT7/	
	69	DAT4/		70	DAT5/	
	71	DAT2/		72	DAT3/	
	73	DAT0/		74	DAT1/	
POWER SUPPLIES	75	GND	Signal GND	76	GND	Signal GND
	77		Reserved	78		Reserved
	79	-12V	-12Vdc	80	-12V	-12Vdc
	81	+5V	+5Vdc	82	+5V	+5Vdc
	83	+5V	+5Vdc	84	+5V	+5Vdc
	85	GND	Signal GND	86	GND	Signal GND

All Mnemonics © Intel Corporation 1978

APPENDIX A (Continued)

P2 CONNECTOR PIN ASSIGNMENT OF OPTIONAL BUS SIGNALS

(COMPONENT SIDE)			(CIRCUIT SIDE)		
PIN	MNEMONIC	DESCRIPTION	PIN	MNEMONIC	DESCRIPTION
1	GND	Signal GND	2	GND	Signal GND
3	5 VB	+5V Battery	4	5 VB	+5V Battery
5		Reserved	6	VCCPP	+5V Pulsed Power
7	-5 VB	-5V Battery	8	-5 VB	-5V Battery
9		Reserved	10	Reserved	
11	12 VB	+12V Battery	12	12 VB	+12V Battery
13	PFSR/	Power Fail Sense Reset	14	Reserved	
15	-12 VB	-12V Battery	16	-12 VB	-12V Battery
17	PFSN/	Power Fail Sense	18	ACLO	AC Low
19	PFIN/	Power Fail Interrupt	20	MPRO/	Memory Protect
21	GND	Signal GND	22	GND	Signal GND
23	+15V	+15V	24	+15V	+15V
25	-15V	-15V	26	-15V	-15V
27	PAR1/	Parity 1	28	HALT/	Bus Master HALT
29	PAR2/	Parity 2	30	WAIT/	Bus Master WAIT STATE
31			32	ALE	Bus Master ALE
33			34	Reserved	
35			36	Reserved	
37			38	AUX RESET/	Reset switch
39			40		
40			42		
43	Reserved		44		
45			46		
47			48		
49			50	Reserved	
51			52		
53			54		
55			56		
57			58		
59			60		
Notes:					
1. PFIN, on slave modules, if possible, should have the option of connecting to INT0/ on P1.					
2. All undefined pins are reserved for future use.					
All Mnemonics © Intel Corporation 1978					

APPENDIX B

BUS TIMING SPECIFICATIONS SUMMARY

Parameter	Description	Minimum	Maximum	Units
t_{BCY}	Bus Clock Period	100	D.C.	ns
t_{BW}	Bus Clock Width	$0.35 t_{BCY}$	$0.65 t_{BCY}$ (Not Restricted)	ns
t_{SKEW}	BCLK/skew		3	ns
t_{PD}	Standard Bus Propagation Delay		3	ns
t_{AS}	Address Set-Up Time (at Slave Board)	50		ns
t_{DS}	Write Data Set Up Time	50		ns
t_{AH}	Address Hold Time	50		ns
t_{DHW}	Write Data Hold Time	50		ns
t_{DXL}	Read Data Set Up Time To XACK	0		ns
t_{DHR}	Read Data Hold Time	0	65	ns
t_{XAH}	Acknowledge Hold Time	0	65	ns
t_{XACK}	Acknowledge Time	0	8	μs
t_{CMD}	Command Pulse Width	100	9.5	ns
t_{ID}	Inhibit Delay	0	100 (Recommend < 100 ns)	ns
t_{XACKA}	Acknowledge Time of an Inhibited Slave	$t_{IAD} + 50$ ns	1500	ns
t_{XACKB}	Acknowledge Time of an Inhibiting Slave	1.5	8	μs
t_{IAD}	Acknowledge Disable from Inhibit (An internal parameter on an inhibited slave; used to determine t_{XACKA} Min.)	0	100 (arbitrary)	ns
t_{AIZ}	Address to Inhibits High Delay		100	ns
t_{INTA}	INTA/ Width	250		ns
t_{CSEP}	Command Separation	100		ns

APPENDIX B (Continued)
BUS TIMING SPECIFICATIONS SUMMARY

Parameter	Description	Minimum	Maximum	Units
tBREQL	↓BCLK/ to BREQ/ Low Delay	0	35	ns
tBREQH	↓BCLK/ to BREQ/ High Delay	0	35	ns
tBPRNS	BPRN/ to ↓BCLK/ Setup Time	22		ns
tBUSY	BUSY/ delay from ↓BCLK/	0	70	ns
tBUSYS	BUSY/ to ↓BCLK/ Setup Time	25		ns
tBPRO	↓BCLK/ to BPRO/ (CLK to Priority Out)	0	40	ns
tBPRNO	BPRN/ to BPRO/ (Priority In to Out)	0	30	ns
tCBRO	↓BCLK/ to CBRQ/ (CLK to Common Bus Request)	0	60	ns
tCBRQS	CBRQ/ to ↓BCLK/ Setup Time	35		ns
tXCD	XACKI to CommandI Delay	0	1500	ns
tBSYO	CBRQ/I and BUSY/I to BUSY/I	—	12	μs
tCCY	C-clock Period	100	110	ns
tCW	C-clock Width	0.35 tCCY	0.65 tCCY	ns
tINIT	INIT/ Width	5		ms
tINITS	INIT/ to MPRO/ Setup Time	100		ns
tPBD	Power Backup Logic Delay	0	200	ns
tPFINW	PFIN/ Width	2.5		ms
tMPRO	MPRO/ Delay	2.0	2.5	ms
tACLOW	ACLO/ Width	3.0		ms
tPFSRW	PFSR/ Width	100		ns
tTOUT	Timeout Delay	5	∞ (D.C.)	ms
tDCH	D.C. Power Supply Hold from ALCO/	3.0		ms
tDCS	D.C. Power Supply Setup to ACLO/	5		ms

APPENDIX C BUS DRIVERS, RECEIVERS, AND TERMINATIONS

Driver 1,3						Receiver 2,3				Termination		
Bus Signals	Location	Type	IOL Min _{ma}	IOH Min _{μa}	CO Max _{pf}	Location	IIL Max _{ma}	IIH Max _{μa}	C _i Max _{pf}	Location	Type	R Units
DAT0/→DATF/ (16 lines)	Masters and Slaves	TRI	16	-2000	300	Masters and Slaves	-0.8	125	18	1 place	Pullup	2.2 KΩ
ADR0/-ADRB/ BHEN/ (21 lines)	Masters	TRI	16	-2000	300	Slaves	-0.8	125	18	1 place	Pullup	2.2 KΩ
MRDC/,MWTC/	Masters	TRI	32	-2000	300	Slaves (Memory; memory- mapped I/O)	-2	125	18	1 place	Pullup	1 KΩ
IORC/.,IOWC/	Masters	TRI	32	-2000	300	Slaves (I/O)	-2	125	18	1 place	Pullup	1 KΩ
XACK/	Slaves	TRI	32	-2000	300	Masters	-2	125	18	1 place	Pullup	510 Ω
INH1/,INH2/	Inhibiting Slaves	OC	16	—	300	Inhibited Slaves (RAM, PROM, ROM, Memory- Mapped I/O)	-2	50	18	1 place	Pullup	1 KΩ
BCLK/	1 place (Master us)	TTL	48	-3000	300	Master	-2	125	18	Mother- board	To +5V To GND	220 Ω 330 Ω
BREQ/	Each Master	TTL	5	-200	60	Central Priority Module	2	50	18	Central Priority Module (not req)	Pullup	1 KΩ
BPRO/	Each Master	TTL	5	-200	60	Next Master in Serial Priority Chain at its BPRN/	-1.6	50	18	(not req)		
BPRN/	Parallel: Central Priority Module Serial: Prev Masters BPRO/	TTL	5	-200	300	Master	-4	100		(not req)		
BUSY/., CBRQ	All Masters	O.C.	20	—	300	All Masters	-2	50	18	1 place	Pullup	1 KΩ
INIT/	Master,	O.C.	32	—	300	All	-2	50	18	1 place	Pullup	2.2 KΩ
CCLK/	1 place	TTL	48	-3000	300	Any	-2	125	18	Mother- board	To +5V To GND	220 Ω 330 Ω
INTA/	Masters	TRI	32	-2000	300	Slaves (Interrupting I/O)	-2	125	18	1 place	Pullup	1 KΩ
INT0/→INT7/ (8 lines)	Slaves	O.C.	16	—	300	Masters	-1.6	40	18	1 place	Pullup	1 KΩ
PFSR/	User's Fron Panel?	TTL	16	-400	300	Slaves, Masters	-1.6	40	18	1 place	Pullup	1 KΩ
PFSN/	Power Back Up Unit	TTL	16	-400	300	Masters	-1.6	40	18	1 place	Pullup	1 KΩ
ACLO	Power Supply	O.C.	16	-400	300	Slaves, Masters	-1.6	40	18	1 place	Pullup	1 KΩ
PFIN/	Power Back- Up Unit	O.C.	16	-400	300	Masters	-1.6	40	18	1 place	Pullup	1 KΩ
MPRO/	Power Back- Up Unit	TTL	16	-400	300	Slaves Masters	-1.6	40	18	1 place	Pullup	1 KΩ

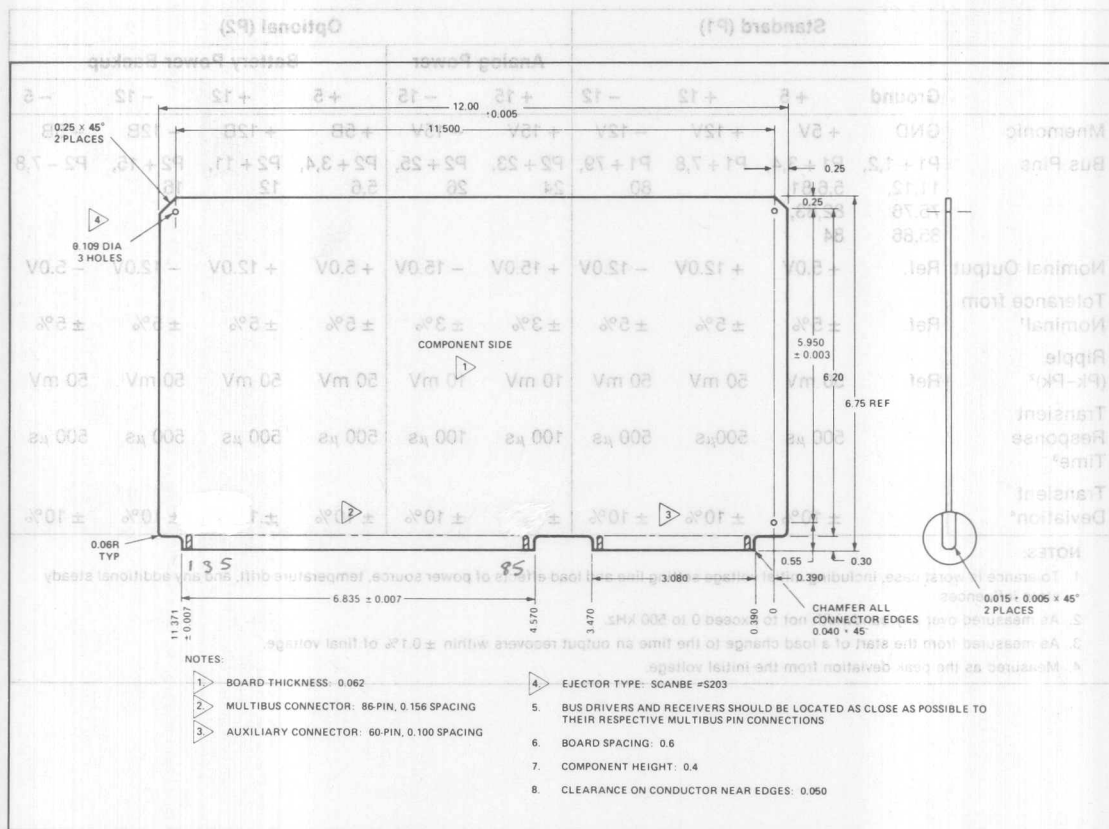
APPENDIX C (Continued)

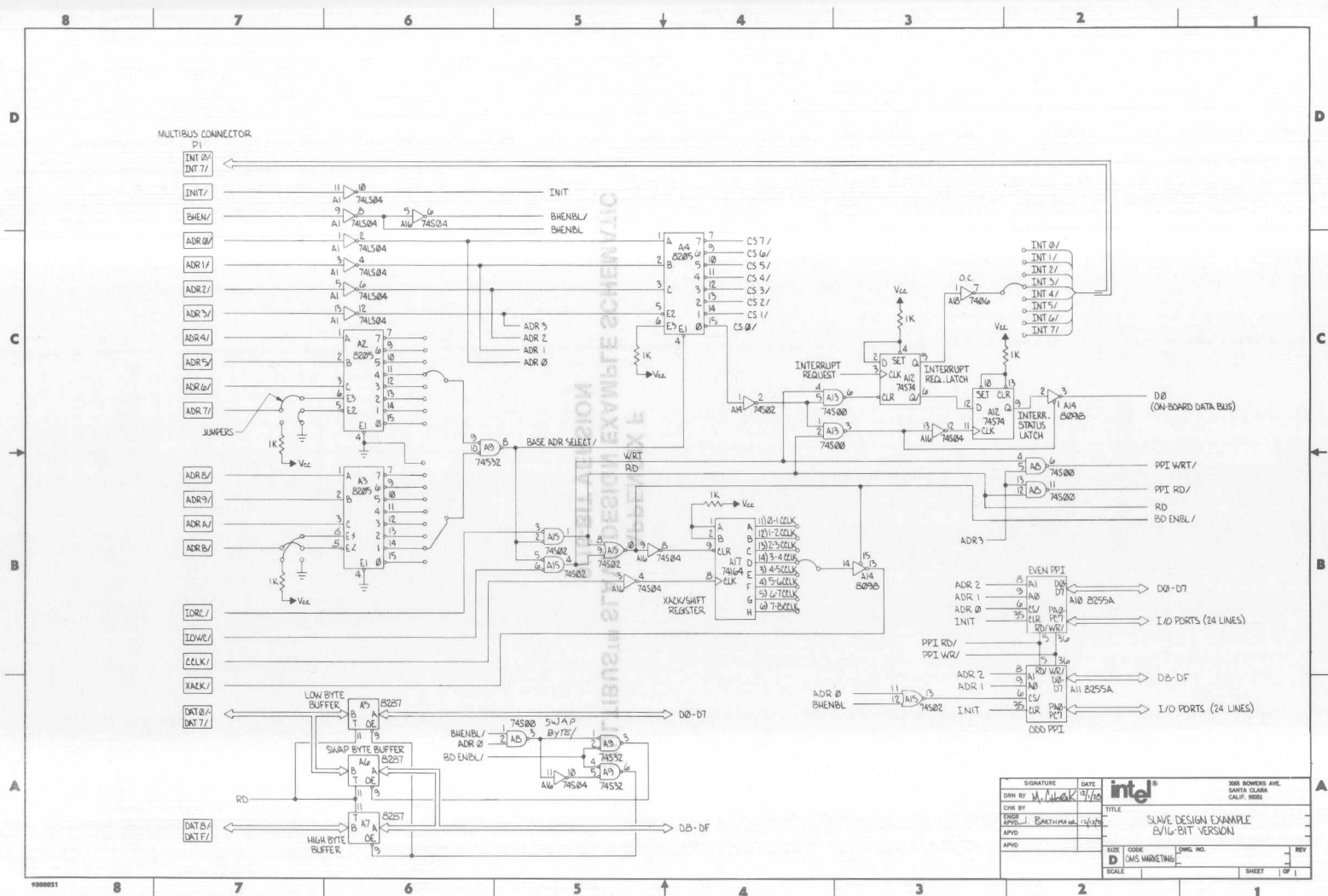
BUS DRIVERS, RECEIVERS, AND TERMINATIONS

Driver 1,3						Receiver 2,3				Termination		
Bus Signals	Location	Type	IOL Min _{ma}	IOH Min _{μa}	CO Max _{pf}	Location	IIL Max _{ma}	I/H Max _{μa}	Ci Max _{pf}	Location	Type	R Units
Aux Reset/	User's Front Panel?	Switch to GND (Note 5)	—	—	—	Masters	-2	50	18	None		
Notes:												
1. Driver Requirements												
IOH = High Output Current Drive												
IOL = Low Output Current Drive												
CO = Capacitance Drive Capability												
TRI = 3-State Drive												
O.C. = Open Collector Driver												
TTL = Totem-pole Driver												
2. Receiver Requirements												
I/H = High Input Current Load												
I/L = Low Input Current Load												
Ci = Capacitive Load												
3. TTL low state must be $\geq -0.5v$ but $\leq 0.8v$ at the receivers												
TTL high state must be $\geq 2.0v$ but $\leq 5.5v$ at the receivers												
4. For the iSBC 80/10 and the iSBC 80/10A use only a 1K pull-up resistor to +5v for BCLK/ and CCLK/ termination.												
5. Recommend a 47Ω resistor in series with switch.												

	Standard (P1)				Optional (P2)							
					Analog Power		Battery Power Backup					
	Ground	+ 5	+ 12	- 12	+ 15	- 15	+ 5	+ 12	- 12	- 5		
Mnemonic	GND	+ 5V	+ 12V	- 12V	+ 15V	- 15V	+ 5B	+ 12B	- 12B	- 5B		
Bus Pins	P1 + 1,2, 11,12, 75,76 85,86	P1 + 3,4, 5,6,81, 82,83, 84	P1 + 7,8	P1 + 79, 80	P2 + 23, 24	P2 + 25, 26	P2 + 3,4, 5,6	P2 + 11, 12	P2 + 15, 16	P2 - 7,8		
Nominal Output	Ref.	+ 5.0V	+ 12.0V	- 12.0V	+ 15.0V	- 15.0V	+ 5.0V	+ 12.0V	- 12.0V	- 5.0V		
Tolerance from Nominal ¹	Ref.	± 5%	± 5%	± 5%	± 3%	± 3%	± 5%	± 5%	± 5%	± 5%		
Ripple (Pk-Pk) ²	Ref.	50 mV	50 mV	50 mV	10 mV	10 mV	50 mV	50 mV	50 mV	50 mV		
Transient Response Time ³		500 μ s	500 μ s	500 μ s	100 μ s	100 μ s	500 μ s	500 μ s	500 μ s	500 μ s		
Transient Deviation ⁴		± 10%	± 10%	± 10%	± 10%	± 10%	± 10%	± 10%	± 10%	± 10%		
NOTES: 1. Tolerance is worst case, including initial voltage setting line and load effects of power source, temperature drift, and any additional steady state influences. 2. As measured over any bandwidth not to exceed 0 to 500 kHz. 3. As measured from the start of a load change to the time an output recovers within $\pm 0.1\%$ of final voltage. 4. Measured as the peak deviation from the initial voltage.												

APPENDIX E
MECHANICAL SPECIFICATIONS





MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC 8/16-BIT VERSION

MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC 8-BIT VERSION



APPLICATION NOTE

AP-43

I. INTRODUCTION	1-81
II. THE iSBC™ 957 SINGLE BOARD COMPUTER	1-81
III. THE iSBC™ 957 PACKAGE	1-82
IV. THE iSBC™ 957-iSBC™ 86V12 MONITOR PROGRAM	1-88
V. MATRIX MULTIPLICATION EXAMPLE	1-92

November 1978

VI. CONCLUSION	1-98
APPENDIX A — iSBC™ 86V12 SIMPLIFIED LOGIC DIAGRAM	1-99
APPENDIX B — PROGRAM LISTINGS FOR EXECUTIONSVEHICLE AND FIND MODULES	1-101
APPENDIX C — PROGRAM LISTING FOR EXECUTIONSVEHICLE MODULE FOR CODE EXPANSION	1-107

Using The iSBC™ 957 Execution Vehicle For Executing 8086 Program Code

Joe Barthmaier
OEM Microcomputer Systems Applications

Using The iSBC™ 957 Execution Vehicle For Executing 8086 Program Code

Contents

I. INTRODUCTION	1-81
II. THE iSBC™ 86/12 SINGLE BOARD COMPUTER	1-81
III. THE iSBC™ 957 PACKAGE	1-85
IV. THE iSBC™ 957-iSBC™ 86/12 MONITOR PROGRAM	1-88
V. MATRIX MULTIPLICATION EXAMPLE	1-92
VI. CONCLUSION	1-98
APPENDIX A — iSBC™ 86/12 SIMPLIFIED LOGIC DIAGRAM	1-99
APPENDIX B — PROGRAM LISTINGS FOR EXECUTION\$VEHICLE AND FIND MODULES	1-101
APPENDIX C — PROGRAM LISTING FOR EXECUTION\$VEHICLE MODULE FOR CODE EXPANSION	1-107

I. INTRODUCTION

The iSBC 957 Intellec—iSBC 86/12 Interface and Execution Package contains the hardware and software required to interface an iSBC 86/12 Single Board Computer with an Intellec Microcomputer Development System. The iSBC 957 package gives the 8086 user the capability to develop software on an Intellec System and then debug this software on an iSBC 86/12 board using a program download capability and an interactive system monitor. The 8086 user has all the capabilities of the Intellec system at his disposal and has the powerful iSBC 86/12 system monitor commands to use for debugging 8086 programs.

The iSBC 86/12 board is an Intel 8086 based processor board which, in addition to the processor, contains 32K bytes of dual port RAM, sockets for up to 16K bytes of ROM/EPROM, a serial I/O port, 24 parallel I/O lines, 2 programmable counters, 9 levels of vectored priority interrupts, and an interface to the MULTIBUS™ system bus. The iSBC 957 package consists of monitor EPROMs for the iSBC 86/12 board, Loader software for the Intellec system, four (4) cable assemblies, assorted line drivers and terminators, and signal adapters. The iSBC 957 package provides the capability of downloading and uploading program and data blocks between an iSBC 86/12 board and an Intellec system. In addition, monitor commands and displays may be input and viewed from the Intellec system console. The iSBC 957 package, when used with the iSBC 86/12 board and an Intellec Microcomputer Development System, provides the user with the capability to edit, compile or assemble, link, locate, download, and interactively debug programs for the 8086 processor. The iSBC 957 package and the iSBC 86/12 board form an excellent "execution vehicle" for users developing software for the 8086 processor regardless of whether the users are 8086 component users or iSBC 86/12 board users. Using the iSBC 957 package 8086 programs may be debugged at the full 5 MHz speed of the processor. The recommended hardware for the execution vehicle is an iSBC 660 system chassis with an 8 card slot backplane and power supply, an iSBC 032 32K byte RAM memory board, the iSBC 957 package, and the iSBC 86/12 board.

This application note will describe how the iSBC 957 package may be used to develop and debug 8086 programs. First a description of the iSBC 86/12 board will be presented. Readers familiar

with the iSBC 86/12 board may want to skip this section. Next follows a detailed description of the iSBC 957 package and the iSBC 86/12 system monitor commands. A program example of a matrix multiplication routine will then be presented. This example will contain both assembly language and PL/M-86 procedures. The steps required to compile, assemble, link, locate and debug the program code will be explained in detail. A typical debugging session using the iSBC 86/12 system monitor will be presented.

II. THE iSBC™ 86/12 SINGLE BOARD COMPUTER

The iSBC 86/12 Single Board Computer, which is a member of Intel's complete line of iSBC 80/86 computer products, is a complete computer system on a single printed-circuit assembly. The iSBC 86/12 board includes a 16-bit central processing unit (CPU), 32K bytes of dynamic RAM, a serial communications interface, three programmable parallel I/O ports, programmable timers, priority interrupt control, MULTIBUS control logic, and bus expansion drivers for interface with other MULTIBUS-compatible expansion boards. Also included is dual port control logic to allow the iSBC 86/12 board to act as a slave RAM device to other MULTIBUS masters in the system. Provision is made for user installation of up to 16K bytes of read only memory. Figure 1 contains a block diagram of the iSBC 86/12 board and in Appendix A is a simplified logic diagram of the iSBC 86/12 board.

Central Processing Unit

The central processor for the iSBC 86/12 board is Intel's 8086, a powerful 16-bit H-MOS device. The 225 sq. mil chip contains 29,000 transistors and has a clock rate of 5MHz. The architecture includes four (4) 16-bit byte addressable data registers, two (2) 16-bit memory base pointer registers and two (2) 16-bit index registers, all accessed by a total of 24 operand addressing modes for complex data handling and very flexible memory addressing.

Instruction Set—The 8086 instruction repertoire includes variable length instruction format (including double operand instructions), 8-bit and 16-bit signed and unsigned arithmetic operators for binary, BCD and unpacked ASCII data, and iterative word and byte string manipulation functions. The instruction set of the 8086 is a functional superset of the 8080A/8085A family and with

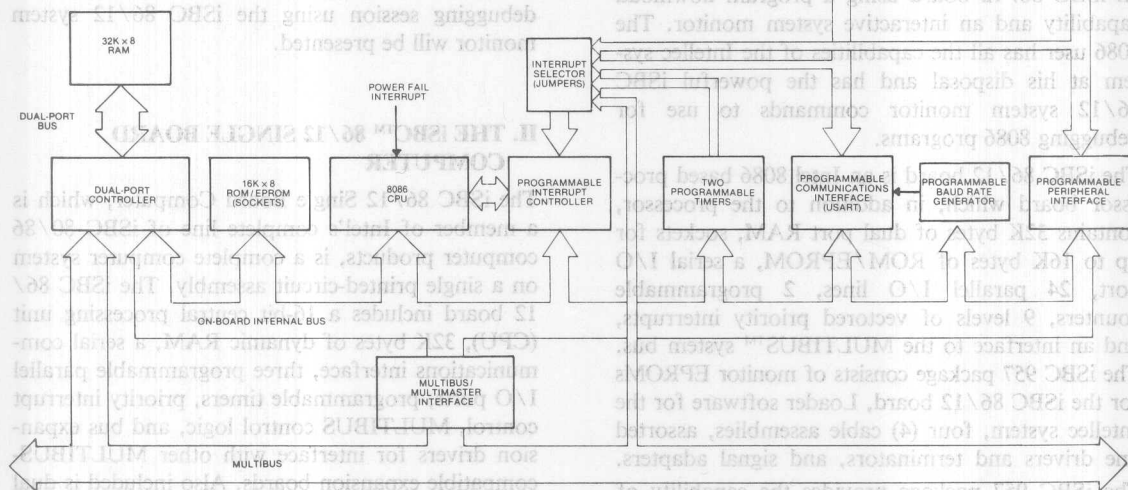


Figure 1. iSBC™ 86/12 Single Board Computer Block Diagram

available software tools, programs written for the 8080A/8085A can be easily converted and run on the 8086 processor.

Architectural Features — A 6-byte instruction queue provides pre-fetching of sequential instructions and can reduce the 1.2 μ sec minimum instruction cycle to 400 nsec by having the instruction already in the queue.

The stack oriented architecture facilitates nested sub-routines and co-routines, reentrant code and powerful interrupt handling. The memory expansion capabilities offer a 1 megabyte addressing range. The dynamic relocation scheme allows ease in segmentation of pure procedure and data for efficient memory utilization. Four segment registers (code, stack, data, extra) contain program loaded offset values which are used to map 16-bit addresses to 20-bit addresses. Each register maps 64K-bytes at a time and activation of a specific register is controlled explicitly by program control and is also selected implicitly by specific functions and instructions.

Bus Structure

The iSBC 86/12 board has an internal bus for communicating with on-board memory and I/O options, a system bus (the MULTIBUS) for referencing additional memory and I/O options, and the dual-port bus which allows access to RAM from the on-board CPU and the MULTIBUS System Bus. Local (on-board) accesses do not require MULTIBUS communication, making the system bus available for use by other MULTIBUS masters (i.e. DMA devices and other single board computers transferring to additional system memory). This feature allows true parallel processing in a multiprocessor environment. In addition, the MULTIBUS interface can be used for system expansion through the use of other 8- and 16-bit iSBC computers, memory and I/O expansion boards.

RAM Capabilities

The iSBC 86/12 board contains 32K-bytes of dynamic read/write memory. Power for the on-board RAM and refresh circuitry may be optionally provided on an auxiliary power bus, and

memory protect logic is included for RAM battery backup requirements. The iSBC 86/12 board contains a dual port controller which allows access to the on-board RAM from the iSBC 86/12's CPU and from any other MULTIBUS master via the system bus. The dual port controller allows 8- and 16-bit accesses from the MULTIBUS System Bus and the on-board CPU transfers data to RAM over a 16-bit data path. Priorities have been established such that memory refresh is guaranteed by the on-board refresh logic and that the on-board CPU has priority over MULTIBUS requests for access to RAM. The dual-port controller includes independent addressing logic for RAM access from the on-board CPU and from the MULTIBUS system bus. The on-board CPU will always access RAM starting at location 00000H. Address jumpers allow on-board RAM to be located starting on any 8K-byte boundary within a 1 megabyte address range for accesses from the MULTIBUS system bus. In conjunction with this feature, the iSBC 86/12 board has the ability to protect on-board memory from MULTIBUS access to any contiguous 8K-byte segments. These features allow multi-processor systems to establish local memory for each processor and shared system (MULTIBUS) memory configurations where the total system memory size (including local on-board memory) can exceed 1 megabyte without addressing conflicts.

EPROM/ROM Capabilities

Four sockets are provided for up to 16K-bytes of non-volatile read only memory on the iSBC 86/12 board. Configuration jumpers allow read only memory to be installed in 2K, 4K, or 8K increments.

On-board ROM is accessed via 16 bit data paths. System memory size is easily expanded by the addition of MULTIBUS compatible memory boards available in the iSBC 80/86 family.

Parallel I/O Interface

The iSBC 86/12 board contains 24 programmable parallel I/O lines implemented using the Intel 8255A Programmable Peripheral Interface. The system software is used to configure the I/O lines in any combination of unidirectional input/output and bidirectional ports.

Therefore, the I/O interface may be customized to meet specific peripheral requirements. In order to take full advantage of the large number of possible I/O configurations, sockets are provided for interchangeable I/O line drivers and terminators. Hence, the flexibility of the I/O interface is further

enhanced by the capability of selecting the appropriate combination of optional line drivers and terminators to provide the required sink current, polarity, and drive/termination characteristics for each application. The 24 programmable I/O lines and signal ground lines are brought out to a 50-pin edge connector that mates with flat, woven, or round cable.

Serial I/O

A programmable communications interface using the Intel 8251A Universal Synchronous/Asynchronous Receiver/Transmitter (USART) is contained on the iSBC 86/12 board. A software selectable baud rate generator provides the USART with all common communication frequencies. The USART can be programmed by the system software to select the desired asynchronous or synchronous serial data transmission technique (including IBM Bi-Sync). The mode of operation (i.e., synchronous or asynchronous), data format, control character format, parity, and baud rate are all under program control. The 8251A provides full duplex, double buffered transmit and receive capability. Parity, overrun, and framing error detection are all incorporated in the USART. The RS232C compatible interface on each board, in conjunction with the USART, provides a direct interface to RS232C compatible terminals, cassettes, and asynchronous and synchronous modems. The RS232C command lines, serial data lines, and signal ground line are brought out to a 26 pin edge connector that mates with RS232C compatible flat or round cable. The iSBC 530 teletypewriter adapter provides an optically isolated interface for those systems requiring a 20 mA current loop. The iSBC 530 adapter may be used to interface the iSBC 86/12 board to teletypewriters or other 20 mA current loop equipment.

Programmable Timers

The iSBC 86/12 board provides three independent, fully programmable 16-bit interval timers/event counters utilizing the Intel 8253 Programmable Interval Timer. Each counter is capable of operating in either BCD or binary modes. Two of these timers/counters are available to the systems designer to generate accurate time intervals under software control. Routing for the outputs and gate/trigger inputs of two of these counters is jumper selectable. The outputs may be independently routed to the 8259A Programmable Interrupt Controller and to the I/O line drivers associated with

the 8255A Programmable Peripheral Interface, or may be routed as inputs to the 8255A chip. The gate/trigger inputs may be routed to I/O terminators associated with the 8255A or as output connections from the 8255A. The third interval timer in the 8253 provides the programmable baud rate generator for the iSBC 86/12 RS232C USART serial port. In utilizing the iSBC 86/12, the systems designer simply configures, via software, each timer independently to meet system requirements. Whenever a given time delay or count is needed, software commands to the programmable timers/event counters select the desired function.

The contents of each counter may be read at any time during system operation with simple read operations for event counting applications, and special commands are included so that the contents can be ready "on the fly".

MULTIBUS™ and Multimaster Capabilities

The MULTIBUS system bus features asynchronous data transfers for the accommodation of devices with various transfer rates while maintaining maximum throughput. Twenty address lines and sixteen separate data lines eliminate the need for address/data multiplexing/demultiplexing logic used in other systems, and allow for data transfer rates up to 5 megawords/sec. A failsafe timer is included in the iSBC 86/12 board which can be used to generate an interrupt if an addressed device does not respond within 6 msec.

Multimaster Capabilities — The iSBC 86/12 board is a full computer on a single board with resources capable of supporting a great variety of OEM system requirements. For those applications requiring additional processing capacity and the benefits of multiprocessing (i.e., several CPUs and/or controllers logically sharing system tasks through communication over the system bus), the iSBC 86/12 board provides full MULTIBUS arbitration control logic. This control logic allows up to three iSBC 86/12 boards or other bus masters, including iSBC 80 family MULTIBUS compatible 8-bit single board computers, to share the system bus in serial (daisy chain) priority fashion, and up to 16 masters to share the MULTIBUS with the addition of an external priority network. The MULTIBUS arbitration logic operates synchronously with a MULTIBUS clock (provided by the iSBC 86/12 board or optionally provided directly from the MULTIBUS System Bus) while data is transferred via a handshake between the master and slave modules. This

allows different speed controllers to share resources on the same bus, and transfers via the bus proceed asynchronously. Thus, transfer speed is dependent on transmitting and receiving devices only. This design prevents slow master modules from being handicapped in their attempts to gain control of the bus, but does not restrict the speed at which faster modules can transfer data via the same bus. The most obvious applications for the master-slave capabilities of the bus are multiprocessor configurations, high speed direct memory access (DMA) operations, and high speed peripheral control, but are by no means limited to these three.

Interrupt Capability

The iSBC 86/12 board provides 9 vectored interrupt levels. The highest level is the NMI (Non-Maskable Interrupt) line which is directly tied to the 8086 CPU. This interrupt cannot be inhibited by software and is typically used for signalling catastrophic events (e.g., power failure).

The Intel 8259A Programmable Interrupt Controller (PIC) provides vectoring for the next eight interrupt levels.

The PIC accepts interrupt requests from the programmable parallel and serial I/O interfaces, the programmable timers, the system bus, or directly from peripheral equipment. The PIC then determines which of the incoming requests is of the highest priority, determines whether this request is of higher priority than the level currently being serviced, and, if appropriate, issues an interrupt to the CPU. Any combination of interrupt levels may be masked, via software, by storing a single byte in the interrupt mask register of the PIC. The PIC generates a unique memory address for each interrupt level. These addresses contain unique instruction pointers and code segment offset values (for expanded memory operation) for each interrupt level. In systems requiring additional interrupt levels, slave 8259A PIC's may be interfaced via the MULTIBUS system bus, to generate additional vector addresses, yielding a total of 65 unique interrupt levels.

Interrupt Request Generation — Interrupt requests may originate from 16 sources. Two jumper selectable interrupt requests can be automatically generated by the programmable peripheral interface.

Two jumper selectable interrupt requests can be automatically generated by the USART when a character is ready to be transferred to the CPU or a character is ready to be transmitted.

A jumper selectable request can also be generated by each of the programmable timers. Eight additional interrupt request lines are available to the user for direct interface to user designated peripheral devices via the system bus, and two interrupt request lines may be jumper routed directly from peripherals via the parallel I/O driver/terminator section.

Power-Fail Control

Control logic is also included to accept a power-fail interrupt in conjunction with the AC-low signal from the iSBC 635 Power Supply or equivalent.

Expansion Capabilities

Memory and I/O capacity may be expanded and additional functions added using Intel MULTIBUS compatible expansion boards. High speed integer and floating point arithmetic capabilities may be added by using the iSBC 310 high speed mathematics unit. Memory may be expanded to 1 megabyte by adding user specified combinations of RAM boards, EPROM boards, or combination boards. Input/output capacity may be increased by adding digital I/O and analog I/O expansion boards. Mass storage capability may be achieved by adding single or double density diskette controllers. Modular expandable backplanes and card-cages are available to support multiboard systems.

III. THE iSBC™ 957 PACKAGE

The iSBC 957 Intellec—iSBC 86/12 Interface and Execution Package extends the software development capabilities of the Intellec Microcomputer Development systems to the Intel 8086 CPU. Programs for the 8086 may be written in PL/M-86 and/or assembly language and compiled or assembled on the Intellec system. These programs may then be downloaded from an Intellec ISIS-II disk file to the iSBC 86/12 board for execution and debug. The programs will execute at the full 5 MHz clock rate of the 8086 CPU with no speed degradation caused by the iSBC 957 hardware or software. Special communication software allows transparent access to the powerful interactive debug commands in the iSBC 86/12 monitor from the Intellec console terminal. These debug commands include single-step instruction execution, execution with breakpoints, memory and register displays, memory searches, comparison of two memory blocks and several other commands. After a debugging session, the debugged program code may be uploaded from the iSBC 86/12 board to an Intellec ISIS-II disk file.

The iSBC 957 Intellec—iSBC 86/12 Interface and Execution Package consists of the following:

- a. Four Intel 2716 EPROMs which contain the system monitor program for the iSBC 86/12 board.
- b. An ISIS-II diskette containing loader software for execution in the Intellec which provides for communications between the user or an Intellec ISIS-II file and the iSBC 86/12 board. Also included on the diskette are a library of routines for system console I/O.
- c. Four cable assemblies used for transmitting commands, code and data between the iSBC 86/12 board and the Intellec system.
- d. An iSBC 530 adapter assembly which converts serial communications signals from current loop to RS232C.
- e. Line drivers and terminators used for the iSBC 86/12 parallel ports.
- f. A small printed circuit board which is plugged into an iSBC 86/12 receiver/terminator socket and is used when program code is downloaded or uploaded using the parallel cable.

iSBC™—Intellec™ Configurations

There are two distinct functional configurations for the iSBC 957 package; one configuration for the Intellec Series II, Models 220 or 230 development systems and another for the Intellec 800 series development systems.

Intellec Series II System Configurations

When used with Intellec Series II Model 220 or 230 systems, a set of cables are used to connect the serial I/O port edge connector on the iSBC 86/12 board and the SERIAL 1 output port on the Intellec system. This configuration is shown in Figure 2. How this system functions is explained in the following paragraphs.

The SERIAL 1 port on the Intellec Series II Model 220 or 230 system is an RS232 port which is designed for use with a data terminal. This port may be used on the Intellec system for interfacing to RS232 devices such as CRT terminals or printers. The serial ports on the iSBC 86/12 board and the Intellec systems are connected as shown in the Figure 2. The flat ribbon cable connected to the iSBC 86/12 board has an edge connector for connecting to the board on one end and a standard RS232 connector on the other end. The second cable, the RS232 Up/Down Load cable, has an RS232 connector on each end. This cable, however,

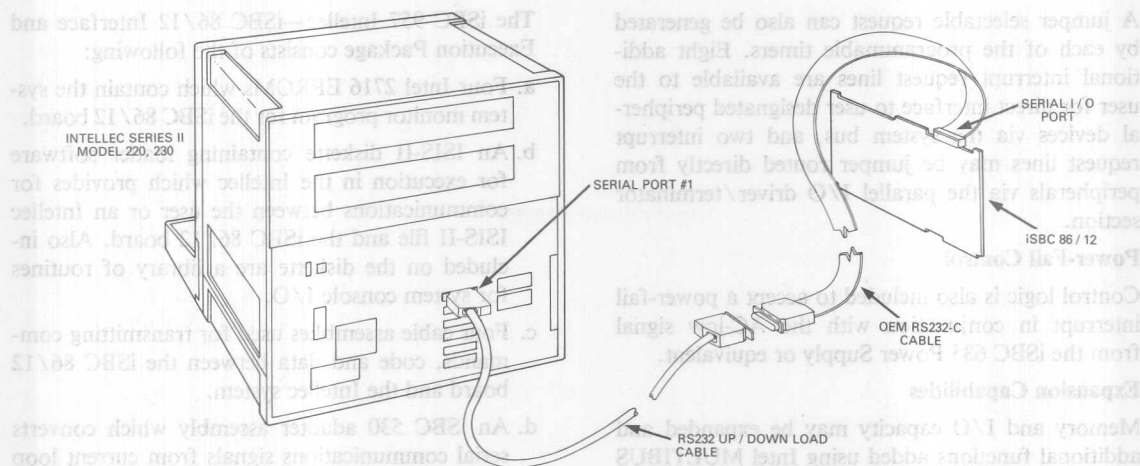


Figure 2. Inteltec™ Series II Model 220, 230—iSBC™ 86/12 Configuration

is not a standard cable with the RS232 signals bussed between identically numbered pins on each of the connectors. The schematic for the cable is shown in Figure 3. Note that the TXD (transmit data) and the RXD (receive data) and the RTS (ready to send) and the CTS (clear to send) signals have been crossed. This is done because both the Inteltec system and the iSBC 86/12 board are configured to act as data sets which are communicating with data terminals. Swapping these signals permits the units to communicate directly with no modifications to the Inteltec or iSBC 86/12 systems themselves.

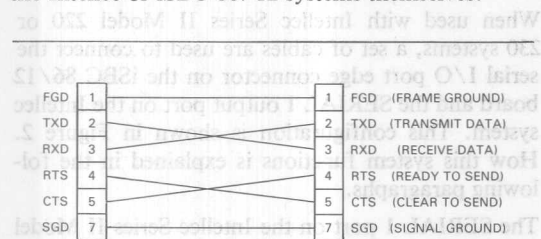


Figure 3. Inteltec™—iSBC™ 86/12 RS232 UP / DOWN LOAD Cable

The software in the Inteltec system accepts characters output from the iSBC 86/12 board through the Inteltec SERIAL 1 port. The software then outputs these characters on the CRT monitor built into the Inteltec Series II Model 220 or 230. In a similar fashion, characters input from the Inteltec key-

board are passed down the serial link to the iSBC 86/12 monitor program. The integrated CRT monitor and keyboard on the Inteltec system then becomes the "virtual terminal" of the iSBC 86/12 monitor program. If this were the only function of the iSBC 957 package, there would be no real benefit to the user. However, when the "virtual terminal" capability is combined with the capability to download and upload program code and data files between the Inteltec ISIS-II file system and the iSBC 86/12 board, a very powerful software development tool is realized. The software in the Inteltec system must examine the commands which are input from the keyboard and in the case of the LOAD and TRANSFER commands (see later sections for details on monitor commands), the software must open and read or write ISIS-II disk files.

Transfer rates using Inteltec Series II Model 220 or 230 system are 9600 baud when transferring hexadecimal object files to or from a disk file and 600 baud when transferring commands between the iSBC 86/12 board and the CRT monitor and keyboard. With a 9600 baud transfer rate, it is possible to load 64K bytes of memory in about four minutes.

Inteltec 800 System Configurations

The iSBC 957 package may be used with the Inteltec 800 system in four different configurations. These four configurations are determined by two

variables. The first variable is whether the iSBC 86/12 board is connected to the Inteltec 800 TTY port or to the Inteltec 800 CRT port. The second variable is whether or not a parallel cable is used for uploading and downloading hexadecimal object files. Figures 4A and 4B illustrate the four configurations.

In Figure 4A, the configuration shows the TTY port of the Inteltec 800 system connected to the iSBC 86/12 serial port using two cables and an iSBC 530 teletypewriter adapter. The TTY port of the Inteltec 800 system is designed for using a teletypewriter as the Inteltec console device. To use this port for communication with the iSBC 86/12 board, the current loop TTY signal must be converted to an RS232 compatible voltage signal. This function is performed by the iSBC 530 adapter.

The cable which connects the Inteltec 800 system to the iSBC 530 adapter performs a function similar to the RS232 Up/Down Load cable described above. A schematic for this cable and all other components of the iSBC 957 package are included with the delivered product.

The transfer rate for both commands and data when the TTY port is connected to the iSBC 86/12 board is 110 baud. This means to download even moderately sized programs would require large amounts of time, several minutes or even hours. However, much faster times may be achieved by using the parallel ports of the iSBC 86/12 board and the Inteltec system for downloading program files. This parallel port used on the Inteltec 800 system is the output port labeled PROM which is normally used with the Universal Prom Pro-

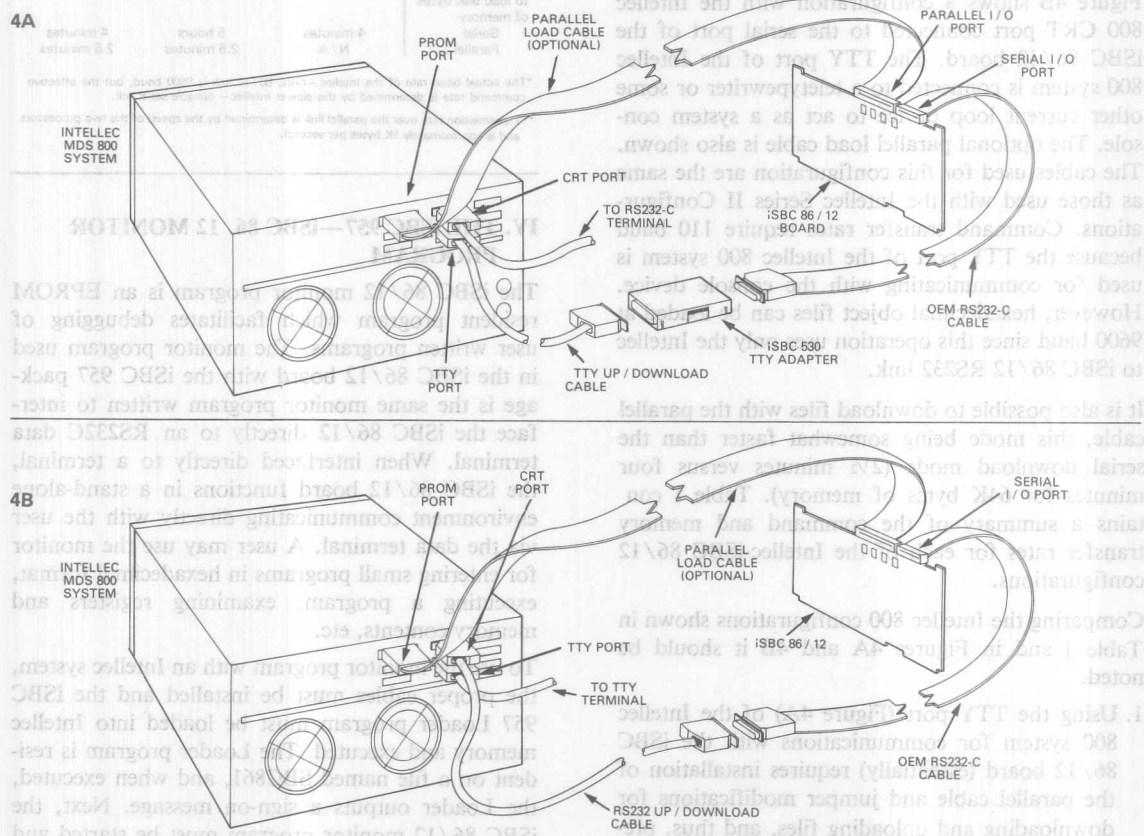


Figure 4A, 4B. Inteltec™ 800—iSBC™ 86/12 Configurations

grammer. A cable is connected between the Intellec PROM port and the parallel I/O port, J1 of the iSBC 86/12 board. Parallel port B of the iSBC 86/12 board is used for the 8-bit byte transfers from the Intellec system to the iSBC 86/12 board, port A is used for the byte transfers from the iSBC 86/12 board to the Intellec system and port C is used for controlling the byte transfers. A special status adapter piggyback board must be inserted into a receiver/terminator socket of the iSBC 86/12 board. This status adapter circuit is required to provide the necessary handshaking signals from the iSBC 86/12 parallel ports to the Intellec PROM port.

The transfer rate achieved when downloading and uploading hexadecimal object files with the parallel cable is approximately 1,000 bytes per second. The time required to load 64K bytes of memory is approximately 2½ minutes.

Figure 4B shows a configuration with the Intellec 800 CRT port connected to the serial port of the iSBC 86/12 board. The TTY port of the Intellec 800 system is connected to a teletypewriter or some other current loop device to act as a system console. The optional parallel load cable is also shown. The cables used for this configuration are the same as those used with the Intellec Series II Configurations. Command transfer rates require 110 baud because the TTY port of the Intellec 800 system is used for communicating with the console device. However, hexadecimal object files can be loaded at 9600 baud since this operation uses only the Intellec to iSBC 86/12 RS232 link.

It is also possible to download files with the parallel cable, this mode being somewhat faster than the serial download mode (2½ minutes versus four minutes for 64K bytes of memory). Table I contains a summary of the command and memory transfer rates for each of the Intellec-iSBC 86/12 configurations.

Comparing the Intellec 800 configurations shown in Table 1 and in Figures 4A and 4B it should be noted:

1. Using the TTY port (Figure 4A) of the Intellec 800 system for communications with the iSBC 86/12 board (essentially) requires installation of the parallel cable and jumper modifications for downloading and uploading files, and thus, prevents the use of the parallel ports for other I/O functions.
2. Using the CRT port (Figure 4B) of the Intellec

800 system for communication with the iSBC 86/12 board provides for a fast serial download capability, thus freeing the parallel ports for other uses. However, this configuration requires a teletypewriter or a CRT capable of accepting a current loop input signal as the Intellec system console.

Table 1
COMMAND AND MEMORY TRANSFER RATES FOR
INTELLEC—iSBC™ 86/12 CONFIGURATIONS

	INTELLEC SERIES II 220/230 SERIAL PORT TO iSBC 86/12	INTELLEC 800 TTY PORT TO iSBC 86/12	INTELLEC 800 CRT PORT TO iSBC 86/12
Effective Command Rate	600 Baud	110 Baud	110 Baud*
Load / Transfer Rate			
Serial	9600 Baud	110 Baud	9600 Baud
Parallel	N / A	1K bytes / sec**	1K bytes / sec**
Approximate Time to load 64K bytes of memory			
Serial	4 minutes	5 hours	4 minutes
Parallel	N / A	2.5 minutes	2.5 minutes

*The actual baud rate of the Intellec—iSBC 86/12 link is 9600 baud, but the effective command rate is determined by the slower Intellec—console serial link.

**Transmission rate over the parallel link is determined by the speed of the two processors and is approximately 1K bytes per second.

IV. THE iSBC 957—iSBC 86/12 MONITOR PROGRAM

The iSBC 86/12 monitor program is an EPROM resident program which facilitates debugging of user written programs. The monitor program used in the iSBC 86/12 board with the iSBC 957 package is the same monitor program written to interface the iSBC 86/12 directly to an RS232C data terminal. When interfaced directly to a terminal, the iSBC 86/12 board functions in a stand-alone environment communicating directly with the user via the data terminal. A user may use the monitor for entering small programs in hexadecimal format, executing a program, examining registers and memory contents, etc.

To use the monitor program with an Intellec system, the proper cables must be installed and the iSBC 957 Loader program must be loaded into Intellec memory and executed. The Loader program is resident on a file named SBC861, and when executed, the Loader outputs a sign-on message. Next, the iSBC 86/12 monitor program must be started and the baud rate of the iSBC 86/12 to Intellec serial communications link must be determined. This is done by pressing the RESET switch on the chassis

Table 2
MONITOR COMMAND LIST

COMMAND	FUNCTION AND SYNTAX
L Load Hex Object File	Loads hexadecimal object file from Intellec into iSBC 86/12 memory using serial (S) or parallel (P) mode. L{S P} <filename> [, <bias addr>] <cr>
T Transfer Hex Object File	Transfers blocks of iSBC 86/12 memory to Intellec as a hex object file using serial (S) or parallel (P) mode. T(X) {S P} <start addr> <end addr> <filename> [, <exec addr>] <cr>
E Exit	Exits the loader program and returns to ISIS. E <cr>
N Single Step	Executes one user program instruction. N[<addr>] [, [<addr>],] * <cr>
G Go	Transfers control of the 8086 CPU to the user program with up to 2 optional breakpoints. G[<start addr>] [, <break 1 addr> [, <break 2 addr>]] <cr>
S Substitute Memory	Displays/modifies memory locations in byte or word format. S(W) <addr> [, [<new contents>],] * <cr>
X Examine/Modify Register	Displays/modifies 8086 CPU registers. X[<reg>] [, [<new contents>],] * <cr>
D Display Memory	Displays contents of a memory block in byte or word format. D(W) <start addr> [, <end addr>] <cr>
M Move	Moves contents of a memory block. M <start addr> <end addr> <destination addr> <cr>
C Compare	Compares two memory blocks. C <start addr> <end addr> <destination addr> <cr>
F Find	Searches a memory block for a byte or word constant. F(W) <start addr> <end addr> <data> <cr>
H Hex Arithmetic	Performs hexadecimal addition and subtraction. H <data 1> <data 2> <cr>
I Port Input	Inputs and displays byte or word data from input port. I(W) <port addr> [,] * <cr>
O Port Output	Outputs byte or word data to output port. O(W) <port addr> <data> [, <data>] * <cr>

Syntax conventions used in the command structure are as follows:

- [A] indicates that "A" is optional
- [A]* indicates one or more optional iterations of "A"
- indicates that "B" is a variable
- {A|B} indicates "A" or "B"
- <cr> indicates a carriage return is entered

Numeric arguments can be expressed as a number, the contents of a register, or the sum or difference of numbers and register contents. Thus, addresses and data can be expressed as follows:

addr ::= [<expr>] <cr>
 expr ::= <number> | <register> | <expr> { + | - } <number> | <expr> { + | - } <register>
 register ::= AX|BX|CX|DX|SP|BP|SI|DI|CS|DS|SS|ES|IP|FL
 number ::= <digit> | <digit> <number>
 digit ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F

Numeric fields within arguments are entered as hexadecimal numbers. The valid range of numerical values is from 0000-FFFF. Larger numbers may be entered, but only the last four digits (or two in the case of byte values) are significant. Leading zeros may be omitted.

An address argument consists of a segment value and an offset value separated by a colon (:). If a segment value is not specified, the default segment value is the CS register value.

containing the iSBC 86/12 board and typing two "U"s on the Intellec console. The ASCII uppercase character U has a binary pattern of alternating ones and zeros, the iSBC 86/12 monitor uses this pattern to determine the baud rate of the serial link. After the baud rate has been determined, the monitor program outputs a sign-on message to the console. An example of loader program execution and monitor program initialization is shown below (user entered characters are underlined).

```
:F1:SBC861
ISIS-II iSBC 86/12 LOADER, Vx.x
(user resets iSBC 86/12 board and types two "U"s)
iSBC 86/12 MONITOR, Vy.y
```

The monitor prompts with a period "." when it is ready for a command. The user can then enter a command file, which consists of a one- or two-character command followed by zero, one, or more arguments. The command may be separated from the first argument by an optional single space; a single comma is required as a delimiter between arguments. The command line is terminated by a carriage return or a comma depending on the command, and no action takes place until the command terminator is sensed. The user can cancel a command before entering the command terminator by pressing any illegal key (e.g., rubout or Control-X).

Table 2 contains a summary of the loader and monitor commands. These commands will not be explained in detail; instead, the next section of the application note will show examples of using these loader and monitor commands. The iSBC 957 User's Guide referenced at the front of this document does, however, contain a complete description of each of the monitor and loader commands.

Table 3 contains a list of the 8086 hardware registers and abbreviations used by the monitor program.

Table 3
8086 CPU REGISTERS

REGISTER NAME	ABBREVIATION
Accumulator	AX
Base	BX
Count	CX
Data	DX
Stack Pointer	SP
Base Pointer	BP
Source Index	SI
Destination Index	DI
Code Segment	CS
Data Segment	DS
Stack Segment	SS
Extra Segment	ES
Instruction Pointer	IP
Flag	FL

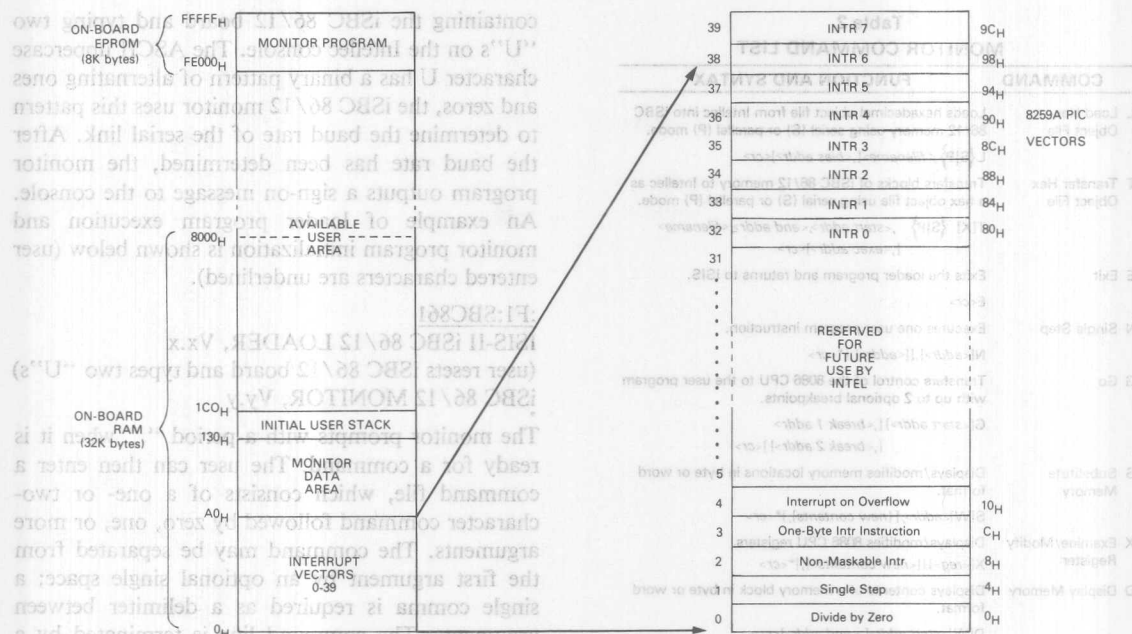


Figure 5. Memory Map of iSBC™ 86/12 Memory With Monitor Program

Figure 5 contains a memory map of the iSBC 86/12 memory with the monitor program. Note that the monitor uses the top 8K bytes of memory for its program code and the first 384 bytes of memory (locations 0 hex to 17F hex) for monitor and user stack, data and interrupt vectors. When the monitor program is reset, the segment registers, the IP and the flags are set to 0; and the SP is set to 01C0H allowing 64 bytes for the user's stack. If 64 bytes is not sufficient for the user's application program, the SP should be set to some other value. The monitor program sets the single-step, one-byte instruction trap and non-maskable interrupt vectors to monitor entry points. The monitor also sets the 8259A Priority Interrupt Controller to fully nested mode with level 0 at the highest priority and all interrupts unmasked. The eight interrupt vector addresses for the 8259A are also set to addresses in the monitor. User programs may change the 8259A interrupt vectors to interrupt service routine addresses within the user programs; it is not necessary for users to program the 8259A chip directly. When an interrupt occurs, control passes to either the monitor or directly to user code depending on the address stored in the vector location. When the monitor responds to an interrupt, it acknowledges the interrupt and displays the interrupt level, CS and IP register values and next instruction byte on

the system console (e.g., I = 3 @ 100:230F F5).

When a user requests a breakpoint with a "G" command, the monitor inserts the single byte instruction trap instructions (INT 3) in the location where the breakpoint is requested. It is also possible for the user to code an INT 3 instruction in his program. When a user coded INT 3 instruction is executed, the monitor will be re-entered and a line with the format @<CS Value>:<IP Value> <Instruction byte> will be displayed (e.g., @1200:3FO2 F5).

Included on the diskette with the Loader program are two libraries containing I/O routines for the console. The library files are named SBCIOS.LIB and SBCIOL.LIB; they contain similar routines. The routines in SBCIOS.LIB are written to be called with intrasegment subroutine calls, a PL/M-86 module compiled with the "small" control generates this type of call. The routines in SBCIOL.LIB are written to be called with intersegment subroutine calls, a PL/M-86 module compiled with either the "medium" or "large" control generates this type of call.

The console input/output routines, CI and CO, contained in the library should be used when performing character input and output on the console. Example PL/M-86 calls to the two routines are:

```

CI: PROCEDURE BYTE EXTERNAL;
END CI;
CO: PROCEDURE (X) EXTERNAL;
  DECLARE X BYTE;
  END CO;

```

```

DECLARE INPUT$CHAR,
        OUTPUT$CHAR BYTE;

```

```

INPUT$CHAR = CI;

```

```

CALL CO(OUTPUT$CHAR);

```

General Comments on Use of the iSBC 957 Package

1. If the iSBC 86/12 board is reset any time after the initial baud rate search, it is not necessary to reload the iSBC 957 Loader program or to download the program code a second time to the iSBC 86/12 board. It is only necessary to re-establish the communications link by typing two "U"s for the baud rate search.
2. The iSBC 86/12 board should not be plugged into an available card slot in an Intellec chassis; a separate chassis should be used. There are at least three reasons for this:
 - a. There is only one RESET signal available on the Intellec system bus. Thus, each processor may not be reset independently. This means that the iSBC 86/12 board cannot be reset without re-booting the ISIS-II operating system and restarting the iSBC 957 Loader.
 - b. The Intellec system uses five of the eight available interrupts on the system bus. This severely restricts the range of interrupts available to the iSBC 86/12 board. Also, the iSBC 86/12 board cannot turn-off the interrupt lamps on the Intellec front panel.
 - c. The iSBC 86/12 board may address up to 1 Megabyte of memory using a 20 bit address. Many Intellec systems contain boards which generate and decode only the low order 16 address bits. For example, the iSBC 016 memory expansion board and the Intellec 800

monitor PROMs only decode 16 address bits. Memory expansion above 64K bytes in these systems is difficult since the boards which decode only 16 bits will force "holes" in the address space above 64K.

3. The iSBC 86/12 board is delivered with two inputs to the 8259A Priority Interrupt Controller connected. Interrupt request 2 (IR2) is connected to the counter 0 output of the 8253 Programmable Interval Timer. IR5 is connected to the INT5/signal of the MULTIBUS System Bus. If these interrupts are not desired, the wire wrap jumpers making the connections should be removed from the iSBC 86/12 board. A particular problem may exist with the counter 0 connection to IR2. If the 8253 counter 0 is not specifically initialized with software, a low frequency square wave output will exist at counter 0's output. This may cause unwanted interrupts when interrupts are enabled by user programs.
4. If the iSBC 86/12 board is used in a system with expansion boards, it is important that the MULTIBUS bus exchange pins be properly jumpered. For example, if the iSBC 86/12 board is used with an iSBC 032 expansion memory board in a system, the BPRN/ MULTIBUS pin for the iSBC 86/12 board should be grounded.

In addition, if any interrupts are used with the iSBC 86/12 board the BPRN/ pin must be grounded. This is true in both single and multiple board systems.

5. Certain user systems require more than one single board computer in the system for performing the functions required by the application. The MULTIBUS System Bus has been specifically designed to permit multiple CPU boards to communicate and to share system resources. However, debugging systems with multiple CPUs has always posed somewhat of a problem. The iSBC 957 package provides a solution to this problem. The serial cable which connects the iSBC 86/12 board to the Intellec system may be removed after the program has been downloaded to the iSBC 86/12 board. A console CRT may then be connected directly to the iSBC 86/12 board and the monitor program may be used to debug the program running on the board. Other iSBC 86/12 boards may also be downloaded from the Intellec system and then switched to their own local terminals. An 8-bit processor board, such as the iSBC 80/30 board, may also be included

in the system, and ICE-85™ may be used for debugging the iSBC 80/30 program concurrently with the iSBC 86/12 programs. Using this scheme, it is possible to debug a system which has several CPU boards by setting breakpoints and using other debugging features on each of the individual CPUs.

V. MATRIX MULTIPLICATION EXAMPLE

To illustrate how the iSBC 957 package can be used to assist in the writing and debugging of 8086 programs on the iSBC 86/12 board, an example program of a matrix multiplication will be presented. The example chosen has been intentionally kept simple and straightforward. The emphasis of this section will be to document the steps required to assemble, compile, link, locate and debug software using an Intellec system, the iSBC 957 package and the iSBC 86/12 board. Part of the example will be written in 8086 assembly language and part in PL/M-86.

The main program is written in PL/M-86. The main program first performs some initialization and the matrix multiplication, then the program calls an assembly language procedure (subroutine), a PL/M-86 procedure and the console output procedure CO supplied in the I/O library on the iSBC 957 diskette. A flow diagram for the example program is shown in Figure 6.

Explanation of the Program Code

The program code is contained in three software modules EXECUTION\$VEHICLE, FIND, and SBCCO. EXECUTION\$VEHICLE contains the main program coded in PL/M-86 and the binary to ASCII conversion procedure BIN\$DEC\$ASC also coded in PL/M-86. The module FIND contains the assembly language procedure FIND\$MX which searches a matrix for its maximum value. The module SBCCO resides in the library of console I/O routines supplied with the iSBC 957 package. The procedure CO will be used from this library.

The program code for the EXECUTION\$VEHICLE and FIND modules will be explained in the following paragraphs. Appendix B contains compilation and assembly listings for the two modules; also contained in Appendix B is a memory and debug map for the linked modules. The listings contain circled reference letters (e.g., (A)) which are referred to by the code description below. The listings in the appendix have been printed on fold-out pages so that they may easily be seen when reading the text.

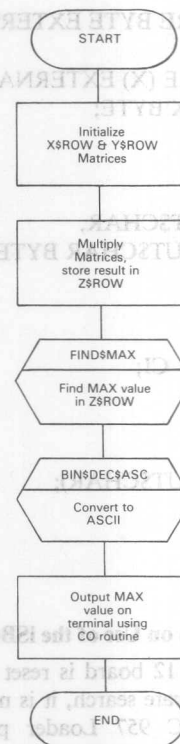


Figure 6.
Flow Diagram of Matrix Multiplication Example

Much of the description given below assumes that the reader is familiar with the PL/M-86 language and compiler, the 8086 assembler, and the link and locate program QRL86. It is recommended that the reader have at least a cursory knowledge of these subjects. The Intel literature for these subjects is listed near the front of this application note.

The EXECUTION\$VEHICLE Module

- (A) The first section of the module includes introductory comments and then statements to declare the matrices, other variables, and procedures used in the program. Note that the matrix dimensions are declared using the literals M, N, and P which are initially set to 6, 5, and 3. Later in this note, other values for M, N, and P will be used.
- (B) The next section of code contains the statements which initialize the two matrices that will be multiplied X\$ROW and Y\$ROW.

As a result of this initialization, the two matrices will contain values as shown in Figure 7.

$$\begin{matrix}
 \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{bmatrix} &
 \begin{bmatrix} 0 & -1 & -2 \\ 0 & -1 & -2 \\ 0 & -1 & -2 \\ 0 & -1 & -2 \\ 0 & -1 & -2 \end{bmatrix} \\
 \text{X\$ROW (6X5)} & \text{Y\$ROW (5X3)}
 \end{matrix}$$

Figure 7.

X\$ROW and Y\$ROW Matrices After Initialization

- (C) The next program section performs the matrix multiplication. The algorithm required to multiply two matrices X and Y, storing the result in a third matrix Z is:

$$Z_{mp} = \sum_{i=1}^n X_{mi} * Y_{ip}$$

Assuming X to be 6X5 matrix and Y a 5X3 matrix then

$$Z_{11} = X_{11}Y_{11} + X_{12}Y_{21} + X_{13}Y_{31} + X_{14}Y_{41} + X_{15}Y_{51}$$

Thus, the upper left term is equal to the sum of the products of the top row of the X matrix times the left column of the Y matrix. The result that is obtained by multiplying the two matrices X\$ROW and Y\$ROW after they are initialized as explained above, is shown in Figure 8.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & -5 & -10 \\ 0 & -10 & -20 \\ 0 & -15 & -30 \\ 0 & -20 & -40 \\ 0 & -25 & -50 \end{bmatrix}$$

Z\$ROW (6X3)

Figure 8. Result of Multiplying the Initialized Matrices X\$ROW and Y\$ROW

- (D) The external assembly language procedure FIND\$MX is called to determine the maximum value in the matrix. The procedure is a typed procedure and returns the maximum value to the calling program which stores it in the integer variable MAX.

- (E) The maximum value is then converted to a six (6) digit ASCII character string by the procedure BIN\$DEC\$ASC. The character string is stored in the array MAX\$ASC\$ARRAY, which contains the sign of the number and five (5) digits for the magnitude.

- (F) Finally, the characters "MAX VALUE =" are output on the system console followed by the 6 ASCII characters containing the maximum value. The PL/M-86 built-in procedure SIZE returns the number of bytes of the array TEXT as a word value. The PL/M-86 built-in procedure SIGNED changes the type of the value from WORD to INTEGER. This is required so that the type of the arguments in the DO statement agree. The console output procedure CO is used to output the characters on the system console.

- (G) Also contained in the module MATRIX.PLM is the binary to ASCII conversion procedure BIN\$DEC\$ASC. The first portion of the code contains the comments explaining the parameters and the calling sequence followed by the declarations. Note that the address of the array where the characters are to be stored is passed to the procedure and that the characters will be stored in the array using based variables. The next section of the code stores either a + or - sign in the first character position of the ASCII array and stores the absolute value of VALUE in the variable TEMP. Finally, the binary value is converted to ASCII using the algorithm explained in the comments. The MOD operator returns the remainder of the division by 10. The UNSIGN built-in procedure is required to change the type of the expression from INTEGER to WORD.

The FIND Module

- (H) The FIND module contains the assembly language procedure FINDMX. The calling sequence and the parameters are explained in the comments at the beginning of the listing. Note that the label FINDMX has been declared PUBLIC so the link program can fill in its address in the CALL statement in the main program of module EXECUTION\$VEHICLE.

- (I) The FIND module will contain three segments: a data segment, a stack segment and a code segment. It will be both convenient and pragmatic to append these three segments to the code, data and stack segments created by the

compiler for the EXECUTION\$VEHICLE module. To accomplish this, the three segments must be given the same SEGMENT and CLASS names as those given these segments by the compiler. The SEGMENT and CLASS names used by the compiler are CODE, DATA, and STACK. The GROUP statements are used to place the segments DATA and STACK in the group DGROUP and the segment CODE in the group CGROUP. These group definitions conform with the group definitions generated by the PL/M-86 compiler when the SMALL size control option is used. A group is a collection of segments which requires less than 64K bytes of memory.

The ASSUME directive informs the assembler that the DS and SS registers will contain the base address of DGROUP and the CS register will contain the base address of CGROUP. This information will be used by the assembler when constructing machine instructions.

J The first segment appearing in the module is the data segment. The order of the segments is arbitrary, although it is recommended that the data segment precede the code segment to minimize forward references to variables which may cause the assembler to generate longer instruction codes. The data segment is declared PUBLIC, aligned on a WORD boundary and given both a segment and class name of DATA. Then follows the contents of the segment. In this particular example, only one word of storage is required. The ENDS directive indicates the end of the segment.

K Next comes the stack segment which is given the segment name of STACK, the combine-type attribute of STACK and the class name of STACK. The combine-type attribute of STACK assures that the stack storage required in this module will be appended to the storage required in the PL/M-86 compiled modules. Two bytes of stack are required by the code in this module, however, the monitor uses 13 words of stack when breakpoints and interrupts are used. Therefore, 14 words are reserved for the stack.

L Finally comes the code segment. The code segment has been given a segment name and class name of CODE and a group name of CGROUP, and has been declared PUBLIC. The alignment attribute of BYTE is specified

since it is desired that the code from this module be appended directly to the code from other modules without gaps between the code modules.

The assembly language code follows next. The code for the procedure must be enclosed between a pair of PROC, ENDP statements. The PROC statement is given the label FINDMX and specified as a NEAR procedure indicating it will be called with a near (intra-segment) CALL instruction and not a far (inter-segment) CALL instruction.

The comments at the beginning of the module and adjacent to the program statements explain the function being performed by the assembly language code.

The SBCCO Module

M The console output procedure CO is contained in the object module SBCCO of the library file SBCIOS.LIB. SBCIOS.LIB is part of the iSBC 957 package I/O libraries. The calling sequence and parameters for CO may be seen in the external procedure declaration in the EXECUTION\$VEHICLE module.

Compiling the EXECUTION\$VEHICLE Module

The EXECUTION\$VEHICLE module is stored on a file named MATRIX.PLM on disk device :F1:. To compile the module, the following command line is used:

— PLM86 :F1:MATRIX.PLM DEBUG

This command line will cause the module stored in the file :F1:MATRIX.PLM to be compiled. The object code generated will be stored in a file with the default name :F1:MATRIX.OBJ and the listing generated will be stored in a file with the default name :F1:MATRIX.LST. To override the default object and listing files, the NOOBJECT and NO-LIST compiler control switches can be used. File names for the listing and object files may also be specified in the command line. The DEBUG compiler control switch causes the compiler to generate extra symbol and line number information which will be used during debugging of the program. A listing of the compiled EXECUTION\$VEHICLE module is contained in Appendix B.

To aid in the debugging of the program, the module was compiled a second time with the following command line:

– PLM86 :F1:MATRIX.PLM NOOBJECT
CODE DEBUG PRINT (:F1:MATRIX.XLS)

This command line specified that no object file is to be created and a listing file should be stored in the file :F1:MATRIX.XLS. The CODE compiler control switch causes the compiler to list the assembly language statements which the compiler has generated for each line of PL/M code. The listing stored in the file MATRIX.XLS is contained in Appendix C.

Assembly of the FIND Module

The assembly language module FIND is stored on a file named FIND.ASM, to assemble this module the following command line is used:

ASM86 :F1:FIND.ASM DEBUG

This command line will cause the FIND module to be assembled with the object code stored in the default file :F1:FIND.OBJ and the listing stored in the default file :F1:FIND.LST. The listing of the assembled FIND module is contained in Appendix B.

Linking and Locating the Object Module

To link and locate the object modules, the QRL86 program will be used. The QRL86 program performs both the linking and the locating of the object modules in a single step. QRL86 is primarily designed for the debugging stages of program development. Some applications may require the extended capabilities of the separate LINK and LOCATE programs when the final link and locate is performed. The command line used to invoke the QRL86 program is:

QRL86 :F1:MATRIX.OBJ, :F1:FIND.OBJ,
SBCIOS.LIB ORIGIN (1000H)

This command line will cause QRL86 to link the code from the three modules and to locate the resultant absolute object module starting at location 1000 hexadecimal. The iSBC 86/12 monitor uses the first 180H bytes of memory for the monitor stack, data and interrupt vectors, 1000H was chosen as a convenient starting address for the program. The absolute object code will be stored in a default file :F1:MATRIX (note no file name extension is used). By default, the memory and debug maps which are generated are stored in the file :F1:MATRIX.MPQ and are contained in Appendix B.

(N) The memory map contains the starting addresses and sizes of the CODE, CONST, DATA, STACK and MEMORY segments of the object module. Note that the start address

for the program is specified as (0100H, 0002H) indicating a CS value of 0100H and an IP value of 0002H or an absolute value of 01002H.

The first two bytes of the code segment contain address values which the code generated by the compiler will use for setting up the DS and SS registers. The memory map shows the code segments from the three modules collected into the group CGROUP. The code segment from the EXECUTION\$VEHICLE module is given the segment and class names of CODE and is put into CGROUP by the PL/M compiler. To assure that the code segment from the FIND module is concatenated with the code segment from the EXECUTION\$VEHICLE module the identical class, segment and group names were specified in the SEGMENT and GROUP statements in the FIND module. Next, the group DGROUP is shown in the memory map. DGROUP contains 4 segments labelled CONST, DATA, STACK and MEMORY. Putting all of these segments in the same group tells the linker that they will all be in the same 64K block of memory. The SMALL size control option of the compiler, which was invoked by default, creates CGROUP, DGROUP, and the segments contained in them.

(P) The debug map contains the memory address of variables, instruction labels and the addresses of each code line of the PL/M-86 module. Notice that the variable storage labels have their addresses specified in the format (DS register value, displacement). For example, the variable TEMP has an address of DS=012AH, displacement = 000CH or an absolute address of 0136H. Instruction labels and line numbers use the format (CS register value, IP register value). Thus, line number six (6) in the module EXECUTION\$VEHICLE has the address CS=0100H, IP=0B5H or 011B5H.

Object to Hex Conversion

Before downloading the program to the iSBC 86/12, the format of the object module must be converted from the absolute object module format which QRL86 creates to a hexadecimal/ASCII representation of the object module. This is done using the program OH86 with the following command line:

OH86 :F1:MATRIX TO :F1:MATRIX.HEX
Downloading and Debugging the Program

The hardware configuration used for debugging the matrix multiplication example program code was

an Intellec Series II Model 230 development system, the iSBC 957 package, an iSBC 86/12 board, and an iSBC 660 system chassis. What follows is the system-user dialog for a typical debugging session.

The first step required is to bootstrap load the ISIS-II operating system by hitting the RESET switch of the Intellec. The Intellec resident loader software is then loaded and executed. Throughout the dialog which follows operator entered characters will be underlined:

```
ISIS-II, V3.4
-SBC861
```

```
ISIS-II iSBC 86/12 LOADER, v1.2
```

To initialize the iSBC 86/12 monitor, the user must hit the RESET switch on the iSBC 660 chassis and type two "U"s on the system console. The monitor program will output a line on the console when it is properly initialized.

```
iSBC 86/12 MONITOR, v1.2
```

The monitor command "X" is typed to check that the monitor is properly operating and to examine the contents of the 8086 registers.

```
.X
AX=0000 BX=0000 CX=0000 DX=0000 SP=01C0 BP=0000 SI=0000
DI=0000 CS=0000 DS=0000 SS=0000 ES=0000 IP=0000 FL=0000
```

To download the hex object file to the iSBC 86/12, the "L" command is used. Because an Intellec Series II Model 230 is being used, a serial download is specified. The hex file name is MATRIX.HEX which is resident on disk device F1:

```
.LS,:F1:MATRIX.HEX
```

The "X" command is used again to examine the CPU registers. Note that the monitor has changed the contents of the CS and IP registers to the value of the starting address of the program.

```
.X
AX=0000 BX=0000 CX=0000 DX=0000 SP=01C0 BP=0000 SI=0000
DI=0000 CS=0100 DS=0000 SS=0000 ES=0000 IP=0002 FL=0000
```

The "D" command is next used to display the first 101 bytes of the program code. Unless another segment register is specified, the display command assumes all addresses specified are relative to the CS register. Thus, the code displayed will be from absolute addresses 1000 through 1100. The program code displayed may be compared with program code generated by the PL/M-86 compiler shown in Appendix C, code line 36.

```
.D0,100
0000 2A 01 FA 2E 8E 16 00 00 BC 08 00 8B EC 16 1F F8
0010 C7 06 8E 00 00 00 81 3E 8E 00 05 00 7E 03 E9 3C
0020 00 C7 06 90 00 00 00 81 3E 90 00 04 00 7E 03 E9
0030 22 00 8B 06 8E 00 89 0A 00 F7 E9 8B 36 90 00 D1
0040 E6 89 C3 8B 0E 8E 00 89 88 10 00 81 06 90 00 01
0050 00 E9 D3 FF 81 06 8E 00 01 00 E9 89 FF C7 06 8E
0060 00 00 00 81 3E 8E 00 04 00 7E 03 E9 40 00 C7 06
0070 90 00 00 00 81 3E 90 00 02 00 7E 03 E9 26 00 8B
0080 06 90 00 F7 D8 50 8B 06 8E 00 89 06 00 F7 E9 8B
0090 36 90 00 D1 E6 89 C3 59 89 88 4C 00 81 06 90 00
00A0 01 00 E9 CF FF 81 06 8E 00 01 00 E9 B5 FF C7 06
00B0 92 00 00 00 81 3E 92 00 02 00 7E 03 E9 8C 00 C7
00C0 06 8E 00 00 00 81 3E 8E 00 05 00 7E 03 E9 72 00
00D0 8B 06 8E 00 89 06 00 F7 E9 8B 36 92 00 D1 E6 89
00E0 C3 C7 80 6A 00 00 00 C7 06 90 00 00 00 81 3E 90
00F0 00 04 00 7E 03 E9 41 00 8B 06 8E 00 89 0A 00 F7
0100 E9
```

The PL/M-86 compiler ends the main program in the EXECUTION\$VEHICLE module with a halt instruction. After execution of the program it is more desirable to return to the monitor. To accomplish this, an INT 3 instruction (code=CC) will be substituted for the halt instruction (code=F4) at the address of 1B4H relative to a CS value of 100H. First the "D" command is used to verify the address of the halt instruction, then the "S" command is used to change the instruction to an INT 3 instruction.

```
.D1B4
01B4 F4
.S1B4, F4- CC
```

To execute the PL/M-86 main program, the "G" command is used. After the "G" is typed, the current contents of the IP are output, followed by the contents of the byte pointed to by the IP. A new value for the IP or breakpoint addresses may be specified before a carriage return <CR> is typed. In this example, only a <CR> is typed.

```
.G 0002- FA
MAX VALUE = -00050
00100:01B5 55
```

The program executes and outputs the maximum value of the matrix calculated. The INT 3 instruction is executed which causes a return to the monitor. The monitor types out an at-sign (@) followed by the CS and IP register values and the first byte of the instruction following the INT 3 instruction.

The "X" command is typed to examine the CPU registers. Note that the program has set both the SS and DS registers to 012A. (012A0H is the address of the DGROUP as shown in the memory map.)

```
.X
AX=0030 BX=0005 CX=000A DX=0000 SP=00D0 BP=00D0 SI=0000
DI=0006 CS=0100 DS=012A SS=012A ES=0000 IP=01B5 FL=F204
```

The three matrices are displayed. Note that a word

display has been specified by using the "DW" Command and that the addresses have been specified relative to the DS register. The addresses of X\$ROW, Y\$ROW, and Z\$ROW may be found in the debug map given by QRL86. Note that the values stored in the matrices are the same as those shown in Figures 8 and 9.

```
.DW DS:10,4A
0010 0000 0000 0000 0000 0000 0001 0001 0001
0020 0001 0001 0002 0002 0002 0002 0002 0003
0030 0003 0003 0003 0003 0004 0004 0004 0004
0040 0004 0005 0005 0005 0005 0005
.DW DS:4C,68
004C 0000 FFFF
0050 FFFE 0000 FFFF FFFE 0000 FFFF FFFE 0000
0060 FFFF FFFE 0000 FFFF FFFE
.DW DS:6A,8C
006A 0000 0000 0000
0070 0000 FFFB FFF6 0000 FFF6 FFEC 0000 FFF1
0080 FFE2 0000 FFEC FFD8 0000 FFE7 FFCE
```

The "G" Command is used to reset the IP register to the start address of the program (002) and to specify a breakpoint at address 0AEH, which is the address of statement 57 of the main program. Statement 57 is the point in the program after the X\$ROW and Y\$ROW matrices have been initialized, but before the matrix multiplication is performed. After the <CR> is typed, the program executes until the breakpoint is encountered. At this point, the monitor outputs a line specifying the number of the breakpoint, the CS and IP values and the first byte of the next instruction to be executed.

```
.G 01B5- 55 002,AE
BR1 00100:00AE C7
```

Next, the single-step capability is used with the "N" command to execute single instructions. At any time, CPU registers may be examined or changed. In this example, the "X" command is used. Execution of succeeding instructions is caused by typing a comma (,).

```
.N 00AE- C7 ,
00B4- 81 ,
00BA- 7E ,
00BF- C7 ,
.X
AX=0018 BX=0018 CX=FFFE DX=0000 SP=00D0 BP=00D0 SI=0004
DI=0006 CS=0100 DS=012A SS=012A ES=0000 IP=00BF FL=F293
.N 00BF- C7 ,
00C5- 81 ,
00CB- 7E ,
```

The contents of the X\$ROW and Y\$ROW matrices are examined and changed with the "SW" (substitute word) command. If a comma (,) is typed after the contents of memory are displayed, then the contents are left unchanged and the next word of memory is displayed. If a value followed by a comma or <CR> is entered, then the contents are changed. If a <CR> is entered, the substitute

sequence is terminated.

```
.SW DS:1A, 0001- 1
001C 0001- 1
001E 0001- 10
.SW DS:5A, FFFF- 1
005C FFFE- 1
005E 0000- 1
0060 FFFF- 64
```

After the matrices are modified, execution is resumed with the "G" command. The max value is output and the INT 3 instruction executed. Finally, the contents of the 3 matrices are displayed.

```
.G 00CB- 7E
MAX VALUE = +00430
00100:01B5 55
.DW DS:10,8C
0010 0000 0000 0000 0000 0000 0001 0001 0010
0020 0001 0001 0002 0002 0002 0002 0002 0003
0030 0003 0003 0003 0003 0004 0004 0004 0004
0040 0004 0005 0005 0005 0005 0005 0000 FFFF
0050 FFFE 0000 FFFF FFFE 0000 FFFF FFFE 0000
0060 0064 FFFE 0000 FFFF FFFE 0000 0000 0000
0070 0000 0051 FFD8 0000 00C0 FFEC 0000 0120
0080 FFE2 0000 0180 FFD8 0000 01E0 FFCE
```

Expanding the Example Program's Memory Requirements

To illustrate how the iSBC 86/12 board may be used for executing 8086 programs which require large amounts of RAM, the example program will be modified. The matrix dimensions of the example will be changed from values of 6, 5 and 3 for the literal symbols of M, N, and P to values of 100, 50, 70. The three matrices will then be of size 100X50, 50X70, and 100X70. The memory required for these matrices is 15.5K words or 31K bytes. The data, constant, stack and memory segments which are contained in the group DGROUP will now comprise almost 32K bytes of memory.

The extra memory requirements will be supplied by using an iSBC 032 board with the iSBC 86/12 board in the iSBC 660 chassis. The iSBC 032 board is a 32K byte RAM board which is compatible with both 8- and 16-bit CPU boards. The base address of the board may be selected anywhere in a 0 to 1 megabyte range on any 16K byte boundary. 8- or 16-bit data transfers may be selected. The iSBC 032 board will be jumpered to respond to addresses in the 512K or 544K address space (20 bit hex address range to 80000H to 87FFFH). This will illustrate the capabilities of the 8086 to access a 20-bit, 1 megabyte address range.

One other modification is required to the program. The magnitude of the numbers which would result from multiplying matrices of this size would greatly exceed the capacity of the 16-bit integer storage, even with the two matrices initialized to the small

values they presently contain. To keep the example simple, the initialization values will be changed so all elements of the X\$ROW matrix are set equal to 2 and all elements of the Y\$ROW matrix are set equal to 3. The result of the multiplication should make all the elements of Z\$ROW equal to 300.

The modified lines of program code are shown below.

```

27 1  DECLARE M LITERALLY '100';
28 1  DECLARE N LITERALLY '50';
29 1  DECLARE P LITERALLY '70';

36 1  DO I = 0 TO (M-1);
37 2  DO J = 0 TO (N-1);
38 3  X$ROW(I).COL(J) = 2;
39 3  END;
40 2  END;

41 1  DO I = 0 TO (M-1);
42 2  DO J = 0 TO (P-1);
43 3  Y$ROW(I).COL(J) = 3;
44 3  END;
45 2  END;

```

The EXECUTION\$VEHICLE module must be re-compiled and then the three program modules must be linked and located using the QRL86 program. Specifying the SEGMENTS option of QRL86, the origin of the CODE segment which is in the group CGROUP is set at 1000H, as in the first example. However, the origin of the CONST, DATA STACK and MEMORY segments which make up the group DGROUP is set at 80000H.

QRL86 :F1:MATRIY.OBJ,:F1:FIND.OBJ,
SBCIOS.LIB SEGMENTS (CODE(1000H),
CONST (80000H), DATA STACK, MEMORY)

The memory map generated by QRL86 shows the CGROUP having a start address of 01000H and the DGROUP having a start address of 80000H.

INVOKED BY:
QRL86 :F1:MATRIY.OBJ,:F1:FIND.OBJ,SBCIOS.LIB &
SEGMENTS (CODE(1000H),CONST(80000H),DATA,STACK,MEMORY)

INPUT MODULES INCLUDED:
:F1:MATRIY.OBJ (EXECUTIONVEHICLE)
:F1:FIND.OBJ (FIND)
SBCIOS.LIB (SBCIO)

RESULT WRITTEN TO :F1:MATRIY(EXECUTIONVEHICLE)
START ADDRESS IS (0100H,0002H)

START	LTH	ALIGN	NAME	CLASS
01000H	298H	G	/GS/ CGROUP	
01000H	210H	W	CODE (EXECUTIONVEHICLE)	CODE
01210H	41H	B	CODE (FIND)	CODE
0125EH	3AH	W	CODE (SBCIO)	CODE
			/GE/ CGROUP	
80000H	7978H	G	/GS/ DGROUP	
80000H	CH	W	CONST (EXECUTIONVEHICLE)	CONST
80000H	0H	W	CONST (SBCIO)	CONST
80000H	792AH	W	DATA (EXECUTIONVEHICLE)	DATA
87936H	2H	W	DATA (FIND)	DATA
87938H	0H	W	DATA (SBCIO)	DATA
87940H	30H	SW	STACK	STACK
87970H	0H	W	MEMORY	MEMORY
87970H	0H	G	/GE/ DGROUP	
87970H	0H	G	??SEGE (FIND)	(NULL)

The object code is then converted to hex format and downloaded to the iSBC 86/12 board. When the program is executed, the maximum value is calculated and output on the console.

```

-SBC861
ISIS-II ISBC 86/12 LOADER, V1.2

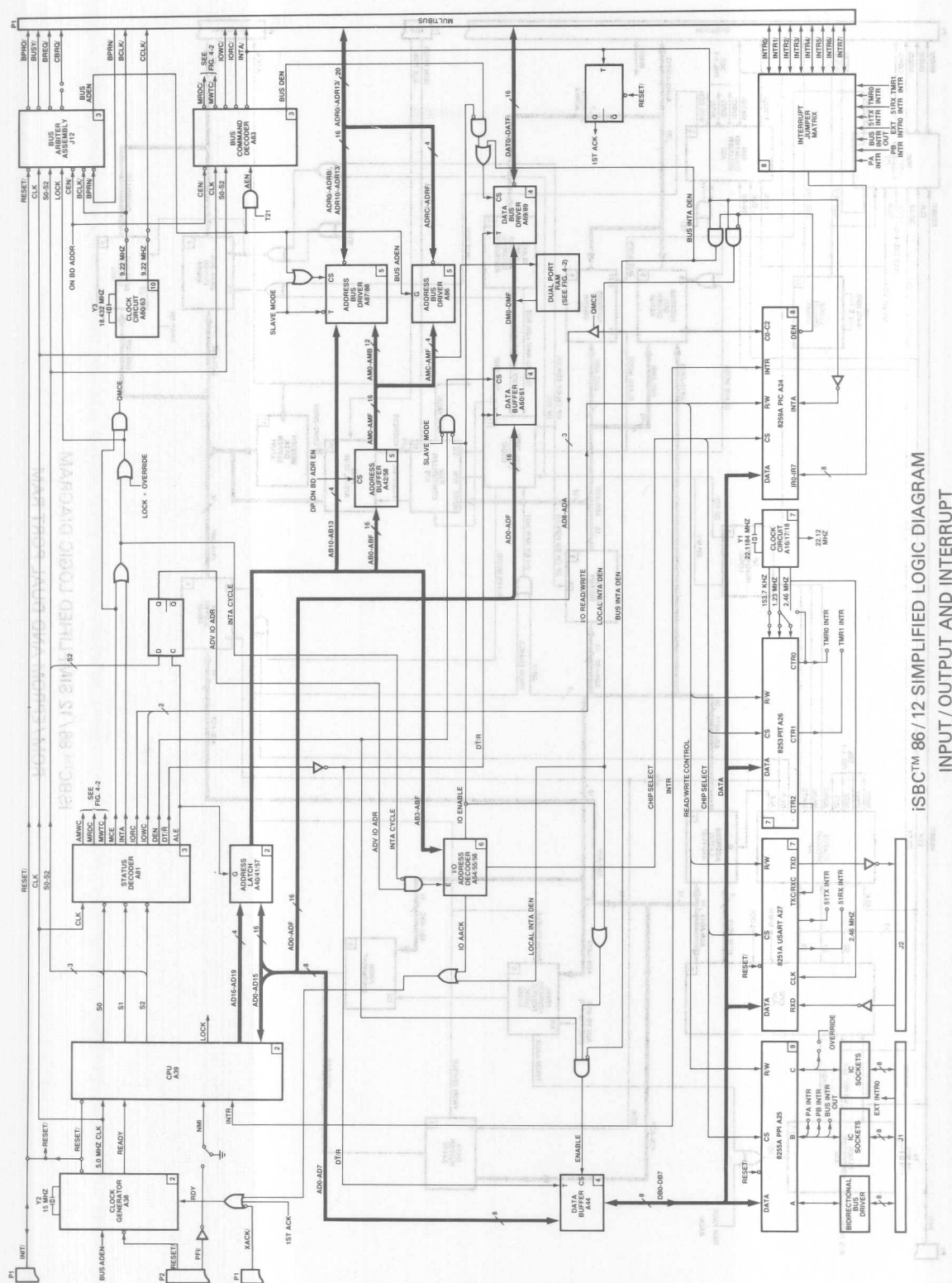
ISBC 86/12 MONITOR, V1.2
.LS,:F1:MATRIY.HEX

.SIAC, F4- CC
.G 0002- FA
MAX VALUE = +00300
@0100:01AD 55

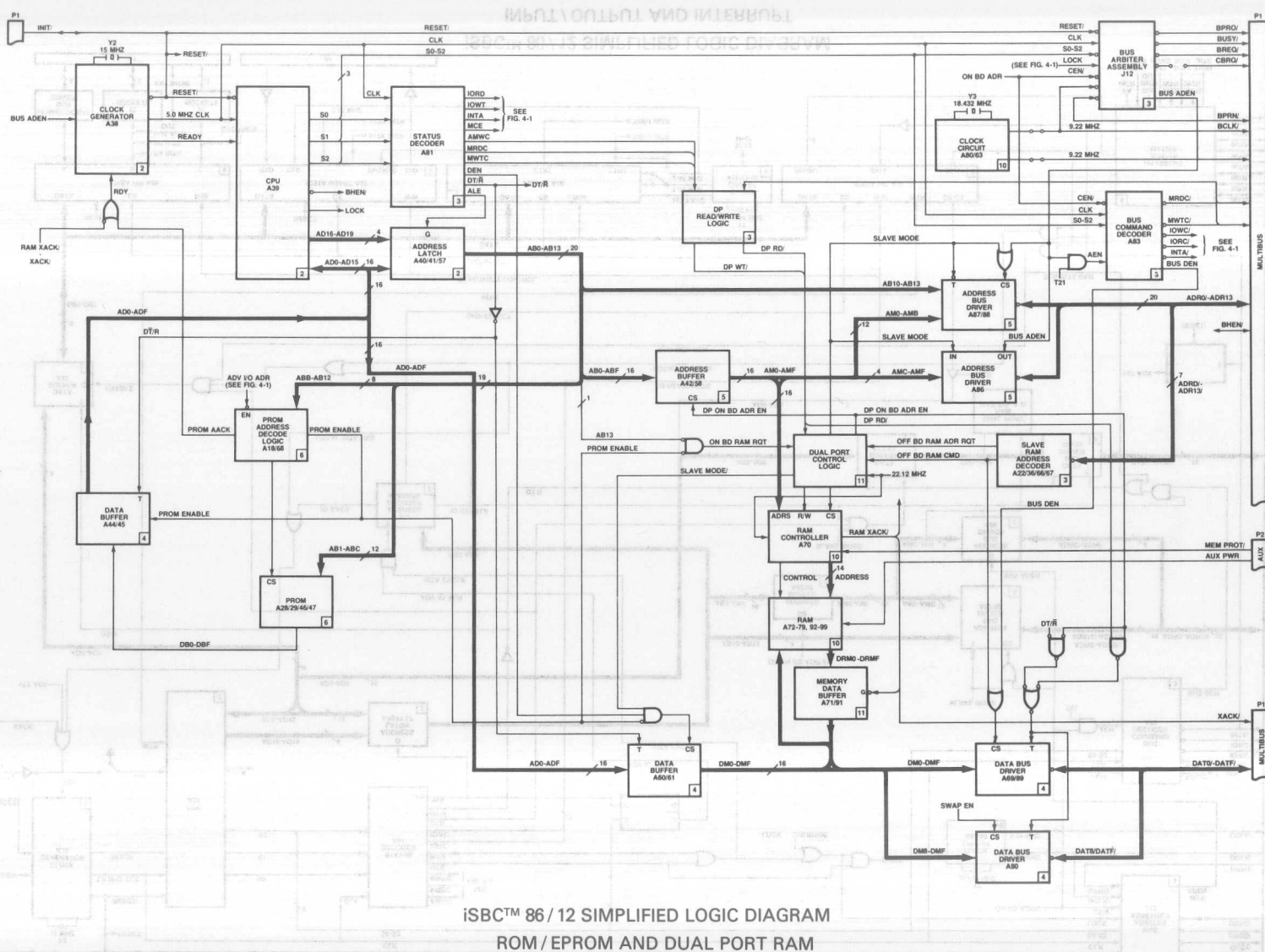
```

VI. CONCLUSION

This application note has described the iSBC 957 Intellec—iSBC 86/12 Interface and Execution Package, and how this package may be used to develop and debug programs for the 8086 processor. First, the iSBC 86/12 single board computer was described, followed by a detailed description of the iSBC 957 package and the iSBC 86/12 system monitor commands. The power and versatility of the iSBC 957 package and monitor commands for developing and debugging programs for the 8086 were illustrated by a program example. In the example a program which consisted of PL/M-86 and assembly language routines was presented. The program code was explained, and the steps required to compile, assemble, link, locate, and debug the program were illustrated. Finally, a typical debugging session using the iSBC 86/12 system monitor which illustrates the powerful capabilities of the monitor was presented.



86/12 SIMPLIFIED LOGIC DIAGRAM INPUT/OUTPUT AND INTERRUPT



ISIS-II PL/M-86 V1.0 COMPILATION OF MODULE EXECUTIONVEHICLE
 OBJECT MODULE PLACED IN :F1:MATRIX.OBJ
 COMPILER INVOKED BY: PLM86 :F1:MATRIX.PLM DEBUG

/* MATRIX MULTIPLICATION EXAMPLE PROGRAM

PL/M-86 MAIN PROGRAM WHICH:

- A) INITIALIZES TWO INTEGER MATRICES
- B) MULTIPLIES THE TWO MATRICES AND STORES THE RESULT IN A THIRD MATRIX
- C) CALLS AN ASSEMBLY LANGUAGE PROCEDURE WHICH SEARCHES THE THIRD MATRIX FOR THE MAXIMUM VALUE
- D) CALLS A PL/M PROCEDURE WHICH CONVERTS THE MAXIMUM VALUE FROM INTEGER TO ASCII
- E) CALLS A PROCEDURE WHICH OUTPUTS THE ASCII CHARACTERS ON THE SYSTEM CONSOLE

*/

1 EXECUTIONVEHICLE:

DO;

/* FINDSMX - EXTERNAL ASSEMBLY LANGUAGE PROCEDURE WHICH SEARCHES A MATRIX FOR THE LARGEST ABSOLUTE MAGNITUDE.

PARAMETERS:

MATRIX\$ADR - ADDRESS OF THE MATRIX TO BE SEARCHED
 ROWS - NUMBER OF ROWS IN THE MATRIX
 COLS - NUMBER OF COLUMNS IN THE MATRIX

*/

2 1 FINDSMX: PROCEDURE (MATRIX\$PTR, ROWS, COLS) INTEGER EXTERNAL;
 3 2 DECLARE (ROWS, COLS) INTEGER;
 4 2 DECLARE MATRIX\$PTR POINTER;
 5 2 END FINDSMX;

/* BINSDECSASC - BINARY TO DECIMAL ASCII CONVERSION PROCEDURE

PARAMETERS:

VALUE - INTEGER VALUE TO BE CONVERTED TO ASCII
 CHARSARRAY\$ADR - ADDRESS OF 6 BYTE ARRAY WHERE ASCII STRING CONTAINING THE VALUE WILL BE STORED

*/

6 1 BINSDECSASC: PROCEDURE (VALUE, CHARSARRAY\$ADR);

7 2 DECLARE (VALUE, TEMP, I) INTEGER;
 8 2 DECLARE CHARSARRAY\$ADR POINTER;
 9 2 DECLARE (CHARSARRAY BASED CHARSARRAY\$ADR) (6) BYTE;

10 2 IF VALUE < 0 THEN
 11 2 DO;
 12 3 CHARSARRAY(0) = '-'; /* SIGN CHARACTER */
 13 3 TEMP = -VALUE;
 14 3 END;
 ELSE
 15 2 DO;
 16 3 CHARSARRAY(0) = '+';
 17 3 TEMP = VALUE;
 18 3 END;
 DO I = 5 TO 1 BY -1;
 20 3 CHARSARRAY(I) = UNSIGN(TEMP MOD 10) + 30H;
 21 3 TEMP = TEMP/10;
 /* ASCII CHARACTERS 30 THRU 39 HEX REPRESENT THE DIGITS 0 THRU 9. THUS
 TO CONVERT AN INTEGER TO ASCII REPEATED DIVISIONS BY 10 AND ADDING
 THE REMAINDER TO 30 HEX WILL ACCOMPLISH THE CONVERSION */
 22 3 END;
 23 2 END BINSDECSASC;

/* CO - EXTERNAL PROCEDURE TO OUTPUT A CHARACTER TO THE SYSTEM CONSOLE.
 THIS PROCEDURE IS PART OF THE ISBC 957 LIBRARY FOR CONSOLE I/O

PARAMETER:

CHAR - ASCII CHARACTER TO BE OUTPUT ON THE CONSOLE

*/

24 1 CO: PROCEDURE (CHAR) EXTERNAL;
 25 2 DECLARE CHAR BYTE;
 26 2 END CO;

/* MATRIX DIMENSIONS */

27 1 DECLARE M LITERALLY '6';
 28 1 DECLARE N LITERALLY '5';
 29 1 DECLARE P LITERALLY '3';

/* THE THREE MATRICES ARE DECLARED AS ARRAYS OF STRUCTURES. XSROW IS COMPOSED OF M STRUCTURES EACH OF WHICH IS COMPOSED OF N INTEGER ELEMENTS. THUS XSROW MAY BE THOUGHT OF AS A M X N MATRIX. THE MATRIX WILL BE STORED AS A ROW-ORDER MATRIX WITH THE ELEMENTS OF EACH ROW STORED IN ADJACENT MEMORY LOCATIONS. Y\$ROW IS DECLARED AS A N X P MATRIX AND Z\$ROW AS A N X P MATRIX */

30 1 DECLARE XSROW(M) STRUCTURE (COL(N) INTEGER);
 31 1 DECLARE Y\$ROW(N) STRUCTURE (COL(P) INTEGER);
 32 1 DECLARE Z\$ROW(M) STRUCTURE (COL(P) INTEGER);
 33 1 DECLARE (I,J,K,MAX) INTEGER;
 34 1 DECLARE MAX\$ASC\$ARRAY(6) BYTE;
 35 1 DECLARE TEXT(*) BYTE DATA ('MAX VALUE = ');

```

/* INITIALIZE X$ROW SUCH THAT THE FIRST ROW IS SET EQUAL TO 0, THE SECOND
/* ROW EQUAL TO 1, THE THIRD ROW EQUAL TO 2, ETC. */
36 1 DO I = 0 TO (M-1);
37 2 DO J = 0 TO (N-1);
38 3 X$ROW(I).COL(J) = I;
39 3 END;
40 2 END;

/* INITIALIZE Y$ROW SUCH THAT THE FIRST COLUMN IS SET EQUAL TO 0, THE
/* SECOND COLUMN EQUAL TO -1, AND THE THIRD COLUMN EQUAL TO -2. */
41 1 DO I = 0 TO (N-1);
42 2 DO J = 0 TO (P-1);
43 3 Y$ROW(I).COL(J) = -J;
44 3 END;
45 2 END;

/* PERFORM MATRIX MULTIPLICATION */
46 1 DO K = 0 TO (P-1);
47 2 DO I = 0 TO (M-1);
48 3 Z$ROW(I).COL(K) = 0; /* SET Z$ROW ELEMENT TO 0 */
49 3 DO J = 0 TO (N-1); /* SUM THE PRODUCT OF X$ROW ROW TERMS AND Y$ROW COLUMN TERMS */
50 4 Z$ROW(I).COL(K) = Z$ROW(I).COL(K) + (X$ROW(I).COL(J) * Y$ROW(J).COL(K));
51 4 END;
52 3 END;
53 2 END;

54 1 MAX = FINDSMX (Z$ROW, M, P); /* FIND MAX VALUE OF Z$ROW */
55 1 CALL BINSDECSASC (MAX, @MAX$ASCSARRAY); /* CONVERT TO DECIMAL ASCII */

56 1 DO I = 0 TO (SIGNED(SIZE(TEXT)) - 1); /* OUTPUT HEADER TEXT */
57 2 CALL CO(TEXT(I));
58 2 END;

59 1 DO I = 0 TO 5; /* OUTPUT ASCII MAX VALUE */
60 2 CALL CO(MAX$ASCSARRAY(I));
61 2 END;

62 1 END EXECUTION$VEHICLE;

MODULE INFORMATION:
CODE AREA SIZE = 0225H 549D
CONSTANT AREA SIZE = 0000H 12D
VARIABLE AREA SIZE = 0000H 144D
MAXIMUM STACK SIZE = 0000H 8D
137 LINES READ
0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION

ISIS-II MCS-86 ASSEMBLER: ASSEMBLY OF MODULE FIND
OBJECT MODULE PLACED IN: F1:FIND.OBJ
ASSEMBLER INVOKED BY: ASM86 :F1:FIND.ASM DEBUG
LOC OBJ SOURCE

LINE SOURCE
1 NAME FIND
2 PUBLIC FINDMX
3
4
5
6
7
8 ASSEMBLY LANGUAGE PROCEDURE TO FIND THE ELEMENT OF AN INTEGER
9 MATRIX WITH THE LARGEST ABSOLUTE MAGNITUDE. THE VALUE OF THE
10 ELEMENT IS RETURNED IN THE AX REGISTER.
11
12 PL/M CALLING SEQUENCE:
13
14 PARAMETERS:
15
16 ADDR$OFMATRIX - ADDRESS OF THE MATRIX WHICH WILL BE SEARCHED
17
18 $SOFROWS - NUMBER OF ROWS IN THE MATRIX
19
20 $SOFSCOLS - NUMBER OF COLUMNS IN THE MATRIX
21
22 PL/M WILL PASS THE THREE PARAMETERS IN THE CALL TO THIS PROCEDURE ON
23 THE STACK. ON ENTRY TO THE PROCEDURE SP+6 WILL POINT TO THE FIRST
24 PARAMETER (ADDR$OFMATRIX) AND SP+4 AND SP+2 WILL POINT TO THE SECOND
25 AND THIRD PARAMETERS.
26
27 THE PROCEDURE IS A TYPED PROCEDURE WHICH ASSIGNS THE MAXIMUM VALUE
28 IN THE MATRIX TO A VARIABLE (IN THIS CASE MAX$VALUE) IN A PL/M
29 ASSIGNMENT STATEMENT. TO ACCOMPLISH THIS ASSIGNMENT THE VALUE IS
30 RETURNED IN THE AX REGISTER.
31
32 THE ALGORITHM USED IS SIMILAR TO THE FOLLOWING PL/M CODE:
33
34 FOR I = 0 TO ($SOFROWS - 1);
35 FOR J = 0 TO ($SOFSCOLS - 1);
36 IF IABS(MATRIX(I).Y(J)) > IABS(MAX) THEN MAX = MATRIX(I).Y(J);
37 END;
38
39 WHERE IABS(XYZ) REPRESENTS THE ABSOLUTE VALUE OF THE INTEGER XYZ

```


LOC	OBJ	LINE	SOURCE
		40	;
		41	;
		42	DEFINE GROUPS TO CONFORM WITH PL/M-86 CONVENTIONS. DATA, STACK, AND
		43	CODE SEGMENTS WILL BE APPENDED TO THEIR RESPECTIVE SEGMENTS IN THE
		44	PL/M-86 MODULES.
		45	DGROUP GROUP DATA,STACK
		46	CGROUP GROUP CODE
		47	;
		48	INSTRUCT THE ASSEMBLER THAT THE DS, SS, AND CS REGISTERS WILL CONTAIN
		49	THE BASE ADDRESS VALUES FOR THE DGROUP, DGROUP AND CGROUP GROUPS.
		50	ASSUME DS:DGROUP,SS:DGROUP,CS:CGROUP
		51	;
		52	;
		53	*****DATA SEGMENT
		54	;
		55	DATA SEGMENT WORD PUBLIC 'DATA'
		56	MAX DW 0
		57	DATA ENDS
		58	;
		59	*****STACK SEGMENT
		60	;
		61	STACK SEGMENT STACK 'STACK'
		62	DW 14 DUP (0) ;RESERVE 13 WORDS OF STACK FOR MONITOR
			AND 1 WORD FOR FINDMX PROCEDURE
		63	;
		64	STACK ENDS
		65	;
		66	*****CODE SEGMENT
		67	;
		68	CODE SEGMENT BYTE PUBLIC 'CODE'
		69	;
		70	PARAMETERS ON STACK, DISPLACEMENT FROM TOS INCREASED BY TWO DUE TO INITIAL PUSH
		71	NO OF ROWS EQU WORD PTR [BP+6]
		72	NO OF COLS EQU WORD PTR [BP+4]
		73	ADR OF MATRIX EQU WORD PTR [BP+8]
		74	;
		75	FINDMX PROC NEAR ;PROCEDURE DECLARATION
		76	BP PUSH ;SAVE BP REGISTER
		77	MOV BP,SP ;BP POINTS TO PARAMETERS ON STACK
		78	XOR DX,DX ;SET DX = ABS OF CURRENT MAX = 0
		79	MOV DI,DX ;DI = I (ROW INDEX) = 0
		80	MOV SI,DX ;SI = J (COLUMN INDEX) = 0
		81	MOV MAX,DX ;MAX = CURRENT MAX = 0
		82	MOV CX,NO_OF_COLS
		83	SHL CX,1 ;CX = (#OF\$COLS) * 2
		84	;
		85	MOV BX,ADR_OF_MATRIX ;ADR\$OF\$MATRIX PARAMETER
		86	;
		87	ABC: MOV AX,[BX][SI] ;BX POINTS TO FIRST ELEMENT OF A GIVEN ROW
		88	OR AX,AX ;GET ELEMENT OF MATRIX
		89	JNS DEF ;SET FLAGS
		90	NEG AX ;JUMP IF SIGN = 0
		91	DEF: CMP AX,DX ;NEGATE TO FORM POSITIVE NUMBER
		92	JL XYZ ;COMPARE TO CURRENT MAX
		93	MOV DX,AX ;JUMP IF LESS THAN CURRENT MAX
		94	MOV AX,[BX][SI] ;MOVE TO ABS OF CURRENT MAX
		95	MOV MAX,AX ;MOVE MATRIX VALUE TO CURRENT MAX
		96	XYZ: ADD SI,2 ;INCREMENT J INDEX BY TWO
		97	CMP SI,CX ;END OF THIS ROW ??
		98	JB ABC ;IF NO, LOOP BACK FOR NEXT ELEMENT OF THIS ROW
		99	LEA BX,[BX+SI] ;BX = BX + (2 * #OF\$COLS), BX POINTS TO NEXT ROW
		100	MOV SI,0 ;J = 0
		101	INC DI ;I = I + 1
		102	CMP DI,NO_OF_ROWS ;LAST ROW ??
		103	JB ABC ;IF NO, DO THE NEXT ROW
		104	MOV AX,MAX ;RETURN MAX VALUE IN AX REGISTER
		105	POP BP ;RESTORE BP REGISTER
		106	RET 6 ;INCREMENT SP BY 6 AND RETURN TO CALLER
		107	;
		108	FINDMX ENDP
		109	;
		110	CODE ENDS
		111	;
		112	END

SYMBOL TABLE LISTING

NAME	TYPE	VALUE	ATTRIBUTES
??SEG	SEGMENT	SIZE=0000H	PARA PUBLIC
ABC	L NEAR	0015H	CODE
ADR OF MATRIX	V WORD	0008H	[BP]
CGROUP	GROUP		CODE
CODE	SEGMENT	SIZE=0041H	BYTE PUBLIC 'CODE'
DATA	SEGMENT	SIZE=0002H	WORD PUBLIC 'DATA'
DEF	L NEAR	001DH	CODE
DGROUP	GROUP		DATA STACK
FINDMX	L NEAR	0000H	CODE PUBLIC
MAX	V WORD	0000H	DATA
NO OF COLS . .	V WORD	0004H	[BP]
NO OF ROWS . .	V WORD	0006H	[BP]
STACK	SEGMENT	SIZE=001CH	PARA STACK 'STACK'
XYZ	L NEAR	0028H	CODE

ASSEMBLY COMPLETE, NO ERRORS FOUND

ISIS-II QRL-P6, V1.1

INVOKED BY:
QRL66 :F1:MATRIX.OBJ,:F1:FIND.OBJ,SBCIOS.LIB ORIGIN(1000H)

INPUT MODULES INCLUDED:
:F1:MATRIX.OBJ(EXECUTIONVEHICLE)
:F1:FIND.OBJ(FIND)
SBCIOS.LIB(SBCCO)

RESULT WRITTEN TO :F1:MATRIX(EXECUTIONVEHICLE)
START ADDRESS IS (0100H,0002H)

(N)

START	LTH	ALIGN	NAME	CLASS
01000H	2A0H	G	/GS/ CGROUP	
01000H	225H	W	CODE(EXECUTIONVEHICLE)	CODE
01225H	41H	B	CODE(FIND)	CODE
01266H	3AH	W	CODE(SBCCO)	CODE
			/GE/ CGROUP	
012A7H	D0H	G	/GS/ DGROUP	
012A0H	CH	W	CONST(EXECUTIONVEHICLE)	CONST
012ACH	0H	W	CONST(SBCCO)	CONST
012ACH	90H	W	DATA(EXECUTIONVEHICLE)	DATA
0133CH	2H	W	DATA(FIND)	DATA
0133EH	0H	W	DATA(SBCCO)	DATA
01340H	70H	SW	STACK	STACK
01370H	0H	W	MEMORY	MEMORY
			/GE/ DGROUP	
0 370H	0H	G	??SEG(FIND)	(NULL)

DEBUG MAP OF :F1:MATRIX(EXECUTIONVEHICLE)

(P)

MODULE: EXECUTIONVEHICLE	0100H,01E1H	LINE #: 19	0100H,0130H	LINE #: 52
012AH,00D0H SYMBOL: MEMORY	0100H,01FBH	LINE #: 20	0100H,0142H	LINE #: 53
0100H,01B5H SYMBOL: BINDECASC	0100H,0213H	LINE #: 21	0100H,0140H	LINE #: 54
012AH,000CH SYMBOL: TEMP	0100H,021EH	LINE #: 22	0100H,015EH	LINE #: 55
012AH,000EH SYMBOL: I	0100H,0221H	LINE #: 23	0100H,0169H	LINE #: 56
012AH,0010H SYMBOL: XROW	0100H,0002H	LINE #: 36	0100H,017AH	LINE #: 57
012AH,004CH SYMBOL: YROW	0100H,0021H	LINE #: 37	0100H,0185H	LINE #: 58
012AH,006AH SYMBOL: ZROW	0100H,0032H	LINE #: 38	0100H,018EH	LINE #: 59
012AH,008EH SYMBOL: I	0100H,004BH	LINE #: 39	0100H,019FH	LINE #: 60
012AH,0090H SYMBOL: J	0100H,0054H	LINE #: 40	0100H,01AAH	LINE #: 61
012AH,0092H SYMBOL: K	0100H,005DH	LINE #: 41	0100H,01B3H	LINE #: 62
012AH,0094H SYMBOL: MAX	0100H,006EH	LINE #: 42	MODULE: FIN	
012AH,0096H SYMBOL: MAXASCARRAY	0100H,007FH	LINE #: 43	0100H,023AH	SYMBOL: ABC
012AH,0000H SYMBOL: TEXT	0100H,009CH	LINE #: 44	0100H,0242H	SYMBOL: DEF
0100H,01B5H LINE #: 6	0100H,00A5H	LINE #: 45	0100H,0225H	SYMBOL: FINDMX
0100H,01B8H LINE #: 10	0100H,00AEH	LINE #: 46	012AH,009CH	SYMBOL: MAX
0100H,01C2H LINE #: 12	0100H,00BFH	LINE #: 47	0100H,024DH	SYMBOL: XYZ
0100H,01C8H LINE #: 13	0100H,00D0H	LINE #: 48	0100H,0225H	PUBLIC: FINDMX
0100H,01D1H LINE #: 14	0100H,00E7H	LINE #: 49	MODULE: SBCCO	
0100H,01D4H LINE #: 16	0100H,00F8H	LINE #: 50	0100H,0266H	PUBLIC: CO
0100H,01DAH LINE #: 17	0100H,0130H	LINE #: 51		

APPENDIX C

PROGRAM LISTING FOR EXECUTION\$VEHICLE MODULE WITH CODE EXPANSION

PL/M-86 COMPILER EXECUTION\$VEHICLE

ISIS-II PL/M-86 V1.0 COMPILATION OF MODULE EXECUTION\$VEHICLE
NO OBJECT MODULE REQUESTED

COMPILER INVOKED BY: PLM86 :F1:MATRIX.PL DEBUG CODE NOOBJECT PRINT(:F1:MATRIX.XLS)

/* MATRIX MULTIPLICATION EXAMPLE PROGRAM

PL/M-86 MAIN PROGRAM WHICH:

- A) INITIALIZES TWO INTEGER MATRICES
- B) MULTIPLIES THE TWO MATRICES AND STORES THE RESULT IN A THIRD MATRIX
- C) CALLS AN ASSEMBLY LANGUAGE PROCEDURE WHICH SEARCHES THE THIRD MATRIX FOR THE MAXIMUM VALUE
- D) CALLS A PL/M PROCEDURE WHICH CONVERTS THE MAXIMUM VALUE FROM INTEGER TO ASCII
- E) CALLS A PROCEDURE WHICH OUTPUTS THE ASCII CHARACTERS ON THE SYSTEM CONSOLE

*/

1 EXECUTION\$VEHICLE:
DO;

/* FINDSMX - EXTERNAL ASSEMBLY LANGUAGE PROCEDURE WHICH SEARCHES A MATRIX FOR THE LARGEST ABSOLUTE MAGNITUDE.

PARAMETERS:

MATRIX\$ADR - ADDRESS OF THE MATRIX TO BE SEARCHED
ROWS - NUMBER OF ROWS IN THE MATRIX
COLS - NUMBER OF COLUMNS IN THE MATRIX

*/

2 1 FINDSMX: PROCEDURE (MATRIX\$PTR, ROWS, COLS) INTEGER EXTERNAL;
3 2 DECLARE (ROWS, COLS) INTEGER;
4 2 DECLARE MATRIX\$PTR POINTER;
5 2 END FINDSMX;

/* BINSDEC\$ASC - BINARY TO DECIMAL ASCII CONVERSION PROCEDURE
PARAMETERS:

VALUE - INTEGER VALUE TO BE CONVERTED TO ASCII
CHARS\$ARRAY\$ADR - ADDRESS OF 6 BYTE ARRAY WHERE ASCII STRING CONTAINING THE VALUE WILL BE STORED

*/

6 1 BINSDEC\$ASC: PROCEDURE (VALUE, CHARS\$ARRAY\$ADR);
; STATEMENT # 5

BINDECASC PROC NEAR

PUSH BP

MOV BP,SP

7 2 DECLARE (VALUE, TEMP, I) INTEGER;
8 2 DECLARE CHARS\$ARRAY\$ADR POINTER;
9 2 DECLARE (CHARS\$ARRAY BASED CHARS\$ARRAY\$ADR) (6) BYTE;

10 2 IF VALUE < 0 THEN

; STATEMENT # 10

01B8 817E06000 CMP [BP].VALUE,0H

01BD 7C03 JL S+5H

01BF E91200 JMP @1

11 2 DO;

12 3 CHARS\$ARRAY(0) = '-'; /* SIGN CHARACTER */

; STATEMENT # 12

01C2 8B5E04 MOV BX,[BP].CHARS\$ARRAY\$ADR

01C5 C6072D MOV CHARARRAY[BX],2DH

13 3 TEMP = -VALUE;

; STATEMENT # 13

01C8 8B4506 MOV AX,[BP].VALUE

01CB F7D8 NEG AX

01CD 89F6000 MOV TEMP,AX

14 3 END;

; STATEMENT # 14

01D1 E9D000 JMP @2

15 2 ELSE

16 3 DO;

CHARS\$ARRAY(0) = '+';

; STATEMENT # 15

01D4 8B5E04 MOV BX,[BP].CHARS\$ARRAY\$ADR

01D7 C6072B MOV CHARARRAY[BX],2BH

17 3 TEMP = VALUE;

; STATEMENT # 17

01DA 8B4506 MOV AX,[BP].VALUE

01DD 8906000 MOV TEMP,AX

18 3 END;

; STATEMENT # 18

01E1 C7060200000 MOV I,5H

01E7 E9F6000 JMP @5

19 2 DO I = 5 TO 1 BY -1;

; STATEMENT # 19

01EA 81060200FFFF ADD I,0FFFFH

```

                                05:
01F0 813E0200100 CMP I,1H
01F6 7D03 JGE $+5H
01F8 E92600 JMP 04
20 3 CHARSARRAY(I) = UNSIGN(TEMP MOD 10) + 30H;
                                ; STATEMENT # 20
01FB 8B060000 MOV AX,TEMP
01FF B90A00 MOV CX,0AH
0202 31D2 XOR DX,DX
0204 F7F9 IDIV CX
0206 81C23000 ADD DX,30H
020A 8B5E04 MOV BX,[BP].CHARARRAYADR
020D 8B360200 MOV SI,1
0211 8B10 MOV [BX].CHARARRAY[SI],DL
21 3 TEMP = TEMP/10;
                                ; STATEMENT # 21
/* ASCII CHARACTERS 30 THRU 39 HEX REPRESENT THE DIGITS 0 THRU 9. THUS
TO CONVERT AN INTEGER TO ASCII REPEATED DIVISIONS BY 10 AND ADDING
THE REMAINDER TO 30 HEX WILL ACCOMPLISH THE CONVERSION */
0213 8B060000 MOV AX,TEMP
0217 99 CWD
0218 F7F9 IDIV CX
021A 8B060000 MOV TEMP,AX
22 3 END;
                                ; STATEMENT # 22
021E E9C9FF JMP 03
                                04:
23 2 END BIN$DECSASC;
                                ; STATEMENT # 23
0221 5D POP BP
0222 C20400 RET 4H
                                BINDECASC ENDP
/* CO - EXTERNAL PROCEDURE TO OUTPUT A CHARACTER TO THE SYSTEM CONSOLE.
THIS PROCEDURE IS PART OF THE ISBC 957 LIBRARY FOR CONSOLE I/O
PARAMETER:
CHAR - ASCII CHARACTER TO BE OUTPUT ON THE CONSOLE
24 1 CO: PROCEDURE (CHAR) EXTERNAL;
25 2 DECLARE CHAR BYTE;
26 2 END CO;
/* MATRIX DIMENSIONS */
27 1 DECLARE M LITERALLY '6';
28 1 DECLARE N LITERALLY '5';
29 1 DECLARE P LITERALLY '3';
/* THE THREE MATRICES ARE DECLARED AS ARRAYS OF STRUCTURES. X$ROW IS COMPOSED
OF M STRUCTURES EACH OF WHICH IS COMPOSED OF N INTEGER ELEMENTS. THUS
X$ROW MAY BE THOUGHT OF AS A M X N MATRIX. THE MATRIX WILL BE STORED AS
A ROW-ORDER MATRIX WITH THE ELEMENTS OF EACH ROW STORED IN ADJACENT MEMORY
LOCATIONS. Y$ROW IS DECLARED AS A N X P MATRIX AND Z$ROW AS A N X P MATRIX */
30 1 DECLARE X$ROW(M) STRUCTURE (COL(N) INTEGER);
31 1 DECLARE Y$ROW(N) STRUCTURE (COL(P) INTEGER);
32 1 DECLARE Z$ROW(M) STRUCTURE (COL(P) INTEGER);
33 1 DECLARE (I,J,K,MAX) INTEGER;
34 1 DECLARE MAX$ASC$ARRAY(6) BYTE;
35 1 DECLARE TEXT(*) BYTE DATA ('MAX VALUE = ');
/* INITIALIZE X$ROW SUCH THAT THE FIRST ROW IS SET EQUAL TO 0, THE SECOND
ROW EQUAL TO 1, THE THIRD ROW EQUAL TO 2, ETC. */
36 1 DO I = 0 TO (M-1);
                                ; STATEMENT # 36
0002 FA CLI
0003 2E0F160000 MOV SS,CS:00STACK$FRAME
0008 BC0000 MOV SP,00STACK$OFFSET
000B 8BEC MOV BP,SP
000D 16 PUSH SS
000E 1F POP DS
000F FB STI
0010 C70682000000 MOV I,PH
                                06:
0016 813E82000500 CMP I,5H
001C 7E03 JLE $+5H
001E E93C00 JMP 07
37 2 DO J = 0 TO (N-1);
                                ; STATEMENT # 37
0021 C70684000000 MOV J,0H
                                08:
0027 813E84000400 CMP J,4H
002D 7E03 JLE $+5H
002F E92200 JMP 09
38 3 X$ROW(I).COL(J) = I;
                                ; STATEMENT # 38
0032 8B068200 MOV AX,I
0036 B90A00 MOV CX,0AH
0039 F7E9 IMUL CX
003B 8B368400 MOV SI,J
003F D1E6 SHL SI,1
0041 89C3 MOV BX,AX
0043 8B0E8200 MOV CX,I
0047 89880400 MOV [BX].X$ROW[SI],CX
39 3 END;

```



```

; STATEMENT # 39
004B 010684000100 ADD J,1H
0051 E9D3FF JMP @0:

40 2 END;

0054 810682000100 ADD I,1H
005A E9B9FF JMP @7:

/* INITIALIZE Y$ROW SUCH THAT THE FIRST COLUMN IS SET EQUAL TO 0, THE
SECOND COLUMN EQUAL TO -1, AND THE THIRD COLUMN EQUAL TO -2. */
41 1 DO I = 0 TO (N-1);
005D C70682000000 MOV I,0H
0063 813E82000400 CMP I,4H
0069 7E03 JLE S+5H
006B E94000 JMP @13:
DO J = 0 TO (P-1);
006E C70684000000 MOV J,0H
0074 813E84000200 CMP J,2H
007A 7E03 JLE S+5H
007C E92600 JMP @13:
Y$ROW(I).COL(J) = -J;
43 3 ; STATEMENT # 43
007F 8B068400 MOV AX,J
0083 F7D8 NEG AX
0085 50 PUSH AX
0086 8B068200 MOV AX,I
008A B90600 MOV CX,6H
008D F7E9 JLE S+5H
008F 8B368400 MOV SI,J
0093 D1E6 SHL SI,1
0095 89C3 MOV BX,AX
0097 59 POP CX
0098 898400 MOV [BX].Y$ROW[SI],CX
END;
009C 810684000100 ADD J,1H
00A2 E9CFFF JMP @13:

45 2 END;
00A5 810682000100 ADD I,1H
00AB E9B5FF JMP @11:

/* PERFORM MATRIX MULTIPLICATION */
46 1 DO K = 0 TO (P-1);
00AE C70686000000 MOV K,0H
00B4 813E86000200 CMP K,2H
00BA 7E03 JLE S+5H
00BC E98C00 JMP @15:
DO I = 0 TO (M-1);
47 2 ; STATEMENT # 47
00BF C70682000000 MOV I,0H
00C5 813E82000500 CMP I,5H
00CB 7E03 JLE S+5H
00CD E97200 JMP @17:
Z$ROW(I).COL(K) = 0; /* SET Z$ROW ELEMENT TO 0 */
48 3 ; STATEMENT # 48
00D0 8B068200 MOV AX,I
00D4 B90600 MOV CX,6H
00D7 F7E9 JLE S+5H
00DD 8B368600 MOV SI,K
00DF D1E6 SHL SI,1
00E1 89C3 MOV BX,AX
00E1 C7805E000000 MOV [BX].Z$ROW[SI],0H
DO J = 0 TO (N-1); /* SUM THE PRODUCT OF X$ROW ROW TERMS AND Y$ROW COLUMN TERMS */
49 3 ; STATEMENT # 49
00E7 C70684000000 MOV J,0H
00ED 813E84000400 CMP J,4H
00F3 7E03 JLE S+5H
00F5 E94100 JMP @19:
Z$ROW(I).COL(K) = Z$ROW(I).COL(K) + ( X$ROW(I).COL(J) * Y$ROW(J).COL(K) );
50 4 ; STATEMENT # 50
00F8 8B068200 MOV AX,I
00FC B90A00 MOV CX,0AH
00FF F7E9 JLE S+5H
0101 8B368400 MOV SI,J
0105 D1E6 SHL SI,1
0107 50 PUSH AX
0108 8B068400 MOV AX,J
010C B90600 MOV CX,6H
010F F7E9 JLE S+5H
0111 8B368600 MOV DI,K
0115 D1E7 SHL DI,1
0117 89C3 MOV BX,AX
0119 8B814000 MOV AX,[BX].Y$ROW[DI]
011D 5B POP BX
011E F7A80400 JLE [BX].X$ROW[SI]
0122 50 PUSH AX
0123 8B068200 MOV AX,I
0127 F7E9 JLE S+5H
0129 89C3 MOV BX,AX

```

```

51 4      @12B 58      POP      AX      ; 1
      @12C @1815E00    ADD      [BX].ZROW[DI],AX
      END;
      ; STATEMENT # 51
      @130 810682000100    ADD      J,1H
      @135 E9B4FF      JMP      @19:
52 3      @19:      END;
      ; STATEMENT # 52
      @139 810682000100    ADD      I,1H
      @13F E983FF      JMP      @16:
53 2      @17:      END;
      ; STATEMENT # 53
      @142 810686000100    ADD      K,1H
      @148 E969FF      JMP      @15:
54 1      MAX = FINDSMX (@Z$ROW, M, P); /* FIND MAX VALUE OF Z$ROW */
      ; STATEMENT # 54
      @14B B85E00      MOV      AX,OFFSET(ZROW)
      @14E 50      PUSH      AX
      @14F B80600      MOV      AX,6H
      @152 50      PUSH      AX
      @153 B80300      MOV      AX,3H
      @156 50      PUSH      AX
      @157 E80000      CALL     FINDMX
      @15A 89068000      MOV      MAX,AX
55 1      CALL BINSDECASC (MAX, @MAX$ASCARRAY); /* CONVERT TO DECIMAL ASCII */
      ; STATEMENT # 55
      @15E FF368000      PUSH      MAX
      @162 B80A00      MOV      AX,OFFSET(MAXASCARRAY)
      @165 50      PUSH      AX
      @166 E84C00      CALL     BINDECASC
56 1      DO I = 0 TO (SIGNED(SIZE(TEXT)) - 1); /* OUTPUT HEADER TEXT */
      ; STATEMENT # 56
      @169 C70682000000      MOV      I,0H
      @172 020:
      @16F @13E82000B00      CMP      I,0BH
      @175 7E03      JLE      S+5H
      @177 E91400      JMP      @21:
57 2      CALL CO(TEXT(I));
      ; STATEMENT # 57
      @17A 8B1E8200      MOV      BX,I
      @17E FFB70000      PUSH     TEXT[BX]; 1
      @182 E80000      CALL     CO
58 2      @185 810682000100    ADD      I,1H
      @18B E9E1FF      JMP      @21:
      ; STATEMENT # 58
59 1      DO I = 0 TO 5; /* OUTPUT ASCII MAX VALUE */
      ; STATEMENT # 59
      @18E C70682000000      MOV      I,0H
      @192 022:
      @194 813E82000500      CMP      I,5H
      @19A 7E03      JLE      S+5H
      @19C E91400      JMP      @23:
60 2      CALL CO(MAX$ASCARRAY(I));
      ; STATEMENT # 60
      @19F 8B1E8200      MOV      BX,I
      @1A3 FFB78A00      PUSH     MAXASCARRAY[BX]; 1
      @1A7 E80000      CALL     CO
61 2      @1AA 810682000100    ADD      I,1H
      @1B0 E9E1FF      JMP      @23:
      ; STATEMENT # 61
62 1      END EXECUTION$VEHICLE;
      ; STATEMENT # 62
      @1B3 FB      STI
      @1B4 F4      HLT

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0225H      549D
CONSTANT AREA SIZE = 000CH      12D
VARIABLE AREA SIZE = 0090H     144D
MAXIMUM STACK SIZE = 0008H      8D
137 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-86 COMPILATION



APPLICATION NOTE

AP-53

October 1979

Using the iSBC 544™ Intelligent Communications Controller

Steve Verleye
OEM Microcomputer Systems Applications

Using the iSBC 544 Intelligent Communications Controller

Contents

I.	INTRODUCTION	1-113
II.	OVERVIEW	1-113
	Intelligent Slave Architecture	1-113
	The iSBC 544 Board	1-115
III.	HARDWARE CONSIDERATIONS ..	1-115
	Two Mode Operation	1-115
	Dual Port RAM	1-116
	Interrupt Structure	1-117
	Modem and Autocall Interface	1-117
IV.	SOFTWARE CONSIDERATIONS ...	1-117
	Device Programming	1-117
	Master/Slave Protocols	1-118
	Communications Support	1-119
V.	THROUGHPUT ANALYSIS	1-119
	Stand-Alone Throughput	1-119
	Intelligent Slave Throughput	1-121
VI.	APPLICATIONS EXAMPLES	1-124
	A Distributed Control System	1-124
	Design Requirements	1-125
	System Configuration	1-126
	Preliminary Design	1-126
	Summary	1-127
	Terminal Cluster Controller	1-127
	Design Criteria	1-127
	System Configuration	1-128
	Preliminary Design	1-129
VII.	SYSTEM SOFTWARE	1-130
	Data Transfer Primitives	1-130
	Sample Slave Software	1-130
	Sample Master Software	1-135
VIII.	SUMMARY	1-136
	APPENDIX A	1-138
	APPENDIX B	1-140
	APPENDIX C	1-145
	APPENDIX D	1-151

I. INTRODUCTION

As the microcomputer system found its way into more and more demanding applications the need became clear for a new and innovative solution to the old problem of providing timely response to real world events. This need was never clearer than in the field of communications where throughput and response time are the keys to success. The iSBC 544 Intelligent Communications Controller (ICC) is the vanguard of a family of intelligent slave computers that provide a unique and powerful answer to the needs of the microcomputer user.

This application note is intended to introduce the reader to the intelligent slave concept in general and the iSBC 544 board in particular. After a brief overview of the evolution of the concept and the features it provides, the hardware and software aspects of the controller are studied. Following this a summary of various system throughput tests is examined to evaluate the performance of the intelligent slave versus more traditional system architectures. We then study two example applications of the product and relate the earlier discussions to the real world. Finally, some system software is presented that handles all data transfer duties between master single board computers and intelligent slaves on the MULTIBUS system bus. More detailed information on many of the topics covered in this note can be found in the related publications listed in the front-piece.

II. OVERVIEW

Intelligent Slave Architecture

Over the years, component technology has increased at a rapid pace going from discrete components (eg. transistors) to integrated circuits (eg. TTL devices) to programmable peripheral controllers (eg. Intel 8251A Universal Synchronous/Asynchronous Receiver/Transmitter) to fully intelligent slave devices (eg. Intel 8041A Universal Peripheral Interface). At the system level the evolution followed a similar path using the increasing component technology to create more and more powerful system building blocks. The iSBC 508 I/O board used TTL logic to provide digital I/O expansion for iSBC computers. The

iSBC 534 board took advantage of programmable LSI devices to provide a programmable communications expansion board. Now, with the advent of the iSBC 544 Intelligent Communications Controller, a new level of system capability is made possible with the fully intelligent slave controller.

The cornerstone of the intelligent slave architecture is the dual port memory. Through the use of this shared memory space, a fast and efficient protocol can be established to allow for cooperation between master and intelligent slave in solving the needs of the application system. In addition to the shared memory, the CPU on the intelligent slave also has some local RAM and local PROM storage for programs. By using this architecture the advantages of multiprocessing and Direct Memory Access (DMA) controllers are blended together. Unlike DMA controllers, the intelligent slave works totally within its own data space. Therefore, it is not affected by bus traffic nor does it add to this traffic. And, since the on-board CPU gets its instructions from local PROM instead of predefined hard-wired logic or microcode, the user has total flexibility in defining the functions the intelligent slave will assume in the overall system.

Although the contents of an intelligent slave make it look very similar to a single board computer, the assumption of the slave role provides a distinct advantage. By performing duties for a master single board computer, the slave relieves the master of low-level processing duties and at the same time is itself relieved of system responsibilities.

In order to position the iSBC 544 product and outline what features it brings to the application system it is necessary to define the functions involved with communicating data. The three main functional divisions are illustrated in Figure 1. At the lowest level the physical interconnection is maintained. This level involves such standards as RS232C which defines the requirements for transmitting bits from point to point. The data transmission level involves the transfer of bytes and/or blocks of data from devices to computers and from node to node in computer networks. The hardware dependent software such as interrupt service and device polling is

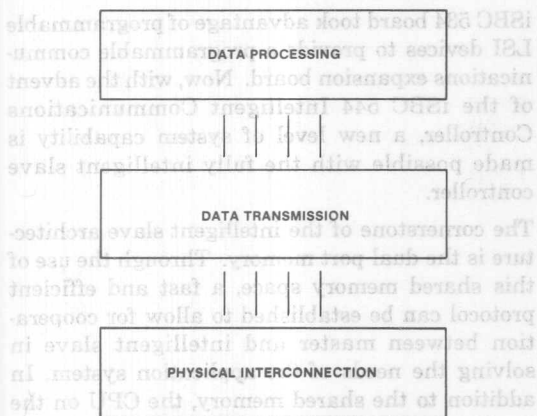


Figure 1. Layering of Communication System Functions

part of this level as are handlers for standard protocols such as SDLC, HDLC, Bisync and X.25 or special purpose schemes and custom protocols.

The highest level performs the actual processing of the data and calls upon the lower levels to move the data from place to place. The application software resides at this level as do some high level software functions such as program to program and process to process communications packages.

Now that we have a map of system functions to guide us, it is possible to gain an understanding of the usefulness of a product like the iSBC 544 Intelligent Communications Controller. If an iSBC 534 board (which contains four USART devices) was included to handle the expansion of serial I/O capacity the mapping of system functions would look like that shown in Figure 2. The four USARTs on the board would handle the physical interconnection but due to the lack of intelligence on the board the master CPU would be burdened with all of the data transmission duties in addition to its real duty, data processing.

When an iSBC 544 board is used in the system, the mapping of system functions is as shown in Figure 3. The physical interconnection is still handled by the USARTs on the board but now the on-board CPU can be programmed to assume the data transmission duties. With an intelligent slave in the system, the master CPU is freed to concentrate on the data processing functions and the end result is that each function in the system is handled in the most efficient manner possible.

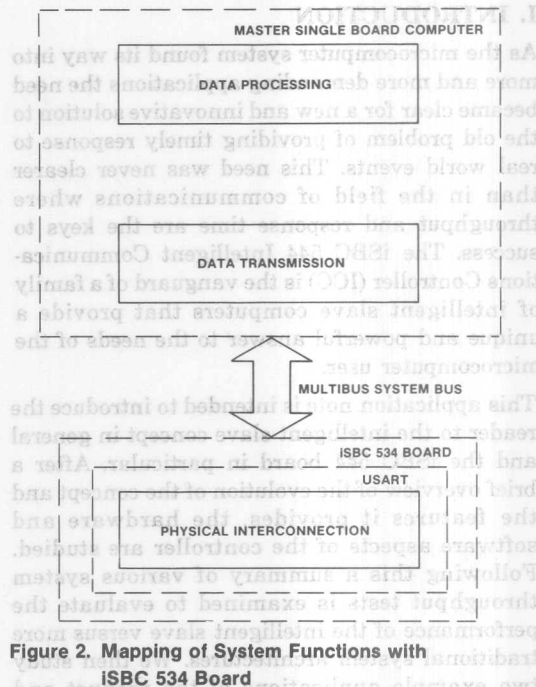


Figure 2. Mapping of System Functions with iSBC 534 Board

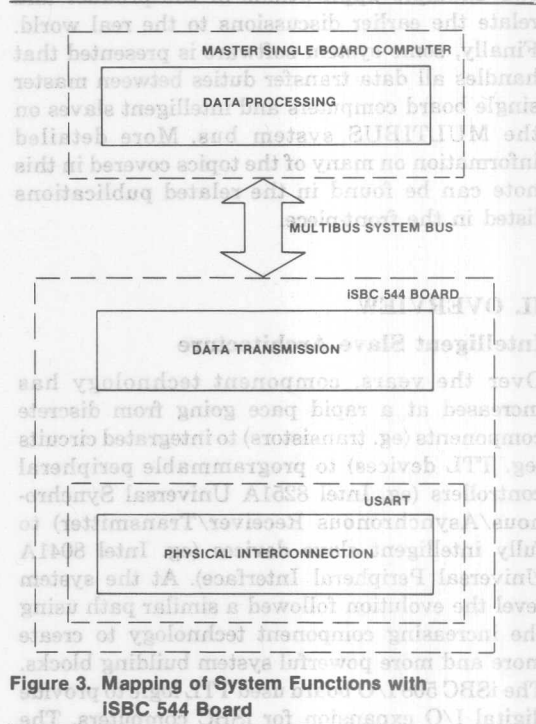


Figure 3. Mapping of System Functions with iSBC 544 Board

The iSBC 544 Board

The iSBC 544 Intelligent Communications Controller contains:

- An Intel 8085A CPU operating at 2.76 MHz.
- Sockets for up to 8K bytes of read only memory (user can choose Intel 2716, 2316E or 2732 devices).
- 16K bytes of dynamic, dual port Random Access Memory (RAM).
- 256 bytes of static local RAM.
- Four Intel 8251A USARTs with programmable baud rates.
- Two Intel 8253 Programmable Interval Timers.
- Intel 8155 parallel interface providing 22 parallel I/O lines and one 14 bit interval timer. Various input and output lines are dedicated to provide an interface to a Bell 801 or equivalent Automatic Call Unit (ACU).
- 8259A Priority Interrupt Controller.

III. HARDWARE CONSIDERATIONS

This section of the application note will focus on the iSBC 544 hardware and will outline the features of the board and its uses. Appendix A contains simplified logic diagrams of the iSBC 544 board which can be referenced in the following discussions.

Two Mode Operation

The iSBC 544 board is capable of operating in one of two modes; 1) intelligent slave and 2) stand-alone communications computer. The mode can either be set with a switch or it can be "toggled" via a software driven flip-flop on the board. In the intelligent slave mode the CPU on the iSBC 544 board operates strictly within its on-board resources. Communications with 8-bit and 16-bit master single board computers is accomplished through the dual port memory. Since the on-board CPU executes code out of its local PROM program storage the system designer is free to define which functions the slave will assume in the system design. As discussed earlier, this could include all or part of the system data transmission duties or could involve application specific duties such as terminal format control, code conversion or terminal input editing.

In the stand-alone mode, the logic on the board disables off board access to the dual port RAM and the bus buffers are used to allow the on-board CPU to access expansion memory and I/O on the MULTIBUS system bus. In this mode the iSBC 544 board drives the bus busy (BUSY/) control line active disallowing any other bus master access to the bus. The stand-alone communications computer is capable of performing all of the functions of the applications system. Referring once again to the diagram of the functions of a communication system, the stand-alone communications computer, with or without system expansion, is responsible for all data transmission and data processing functions. In small applications requiring multiple serial lines the stand-alone iSBC 544 controller is a perfect fit.

In very special circumstances it is possible to share the system bus by toggling the mode set flip-flop between master and slave mode. Figure 4

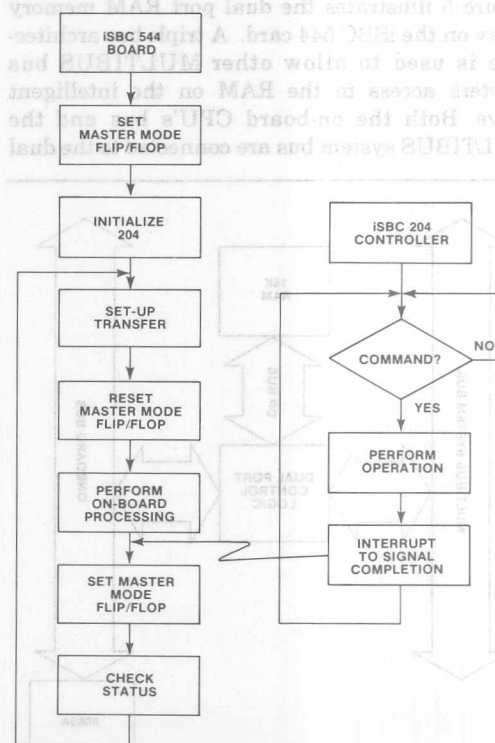


Figure 4. iSBC 544 Controller Running iSBC 204 Disk Controller

shows the flow chart for a routine (code in Appendix B) that makes use of the "software switch" to operate an iSBC 204 Diskette Controller. Using the iSBC 544 board in a system with DMA devices is not recommended except in cases where DMA accesses are short and relatively rare. The use of the CPU for the handling of other system devices could seriously degrade its performance as a communications controller. However, this capability could be extremely useful in a system such as a small message store and forward where the disk traffic is not heavy and including a CPU card just to handle the disk would be wasteful. Use of the "software switch" to share the bus with another iSBC CPU is not advised because of the amount of protocol that would be required to keep the CPUs from interfering with each other on the bus.

Dual Port RAM

Figure 5 illustrates the dual port RAM memory array on the iSBC 544 card. A triple bus architecture is used to allow other MULTIBUS bus masters access to the RAM on the intelligent slave. Both the on-board CPU's bus and the MULTIBUS system bus are connected to the dual

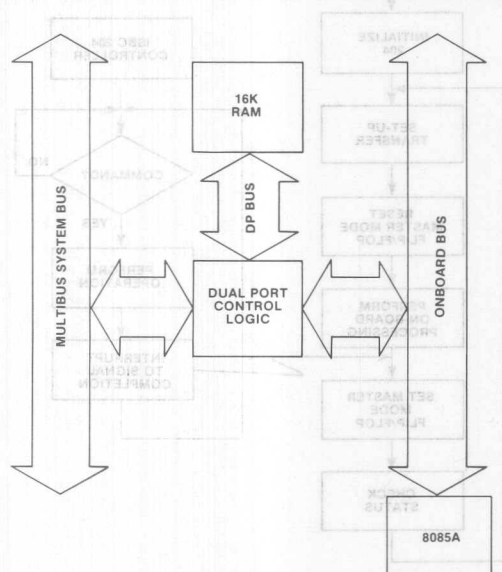


Figure 5. Dual Port Control Logic

port controller. From here the dual port bus is connected to the 16K of dynamic RAM memory. Memory transfer requests from either of the first two busses are handled by the dual port control logic with the on-board CPU being given priority if contention arises. The local CPU is favored so that it is not overly delayed in handling its time critical functions.

The address mapping of the dual port memory on the iSBC 544 is diagrammed in Figure 6. The user can enable access from the MULTIBUS system bus to 0, 4K, 8K or all 16K of the RAM on each iSBC 544 board. The dual port control logic decodes the full 20-bit address and provides an 8-bit data path to the bus. For these reasons the iSBC 544 board is compatible with 8080A, 8085A and 8086 based single board computers. The user can also select the block of addresses on the system bus to which the iSBC 544 RAM will respond.

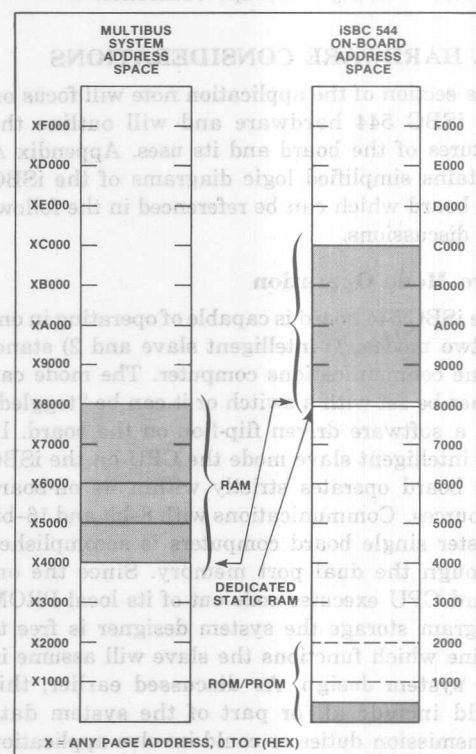


Figure 6. Address Mapping on Dual Port RAM Block

When accessed by the on-board CPU, the dual port RAM always appears at 8000H. If the iSBC 544 board is operating in the stand-alone computer mode, the board is capable of generating the 16-bit bus address supported by the 8085A CPU.

Interrupt Structure

The interrupt structure of the iSBC 544 controller is designed to handle the heavy load imposed by the inherent real-time nature of the communications application. An 8259A Priority Interrupt Controller handles the four receiver and transmitter ready interrupts from the 8251A devices and provides vectored interrupts using one of many available priority schemes. In addition to the eight interrupt sources handled by the 8259 there are various others that can be connected directly to the vectored interrupt inputs on the 8085A (RST 5.5, 6.5, 7.5 and TRAP). One interrupt is generated by the dual port control logic whenever a byte is written into the base address of the dual port memory by an offboard CPU. This interrupt, the flag interrupt, is cleared automatically when the on-board CPU reads the byte and is useful when designing a master-slave protocol since it provides a unique interrupt to each slave in the system.

If the 8251A devices are used to interface to modems the loss of carrier and ring indicator interrupts from all four channels need to be connected to 8085A interrupt request inputs. This is accomplished with four input OR gates tying the eight sources into RST 6.5. The ring indicator and carrier detect lines can also be monitored through a parallel I/O port. This port would be read in a polled system to determine status or could be used along with the OR-tied interrupts to determine which channel is sourcing the current interrupt.

The remaining interrupt sources come from the extra timer/counters and from the MULTIBUS interrupt lines. In addition to receiving interrupts from the bus, the iSBC 544 board has the capability of generating MULTIBUS interrupts using the Serial Output Data (SOD) line on the 8085A CPU.

Modem and Autocall Interface

The iSBC 544 controller uses 8251A and 8155 devices for interface to modems and an autocall

unit respectively. All of the necessary handshaking signals concerned with the modem interface are connected to the 8251A and the carrier detect and ring indicator signals, as previously mentioned, can be connected to interrupt inputs. The 8155 parallel ports are wired as shown in Figure 7. All of the commonly used signals defined in the EIA RS-366 specification for interface to an autocall unit are provided. The software necessary for handling the ACU becomes a simple matter of responding to the ACU requests and sending out the BCD digits representing the number being dialed. In addition to the ACU interface, the 8155 monitors various signal states and provides software reset capabilities for the USARTs and some interrupts.

IV. SOFTWARE CONSIDERATIONS

Software for the iSBC 544 ICC falls into three main categories; device programming, master-slave protocols, and communications support. Each of these three topics is covered in the following section with the aim of defining the software requirements and functions of the iSBC 544 board.

Device Programming

The main sources of the power and flexibility of this product are the programmable LSI devices on the board. The first duty of the on-board software is programming these devices to handle the specific task at hand. To start with, the 8251A USART can be programmed for synchronous or asynchronous operation. In synchronous mode the user specifies even, odd or no parity and either external or internal sync detect with one or two sync characters. In the asynchronous mode the programmer selects the parity, the character length (5, 6, 7 or 8 data bits), the framing control (1, 1½ or 2 stop bits) and the baud rate scaling factor (input clock frequency divided by 1, 16 or 64).

The 8253 Programmable Interval Timers provide the receiver and transmitter clocks for the USARTs and, along with the 8251A baud rate scaling factor, are programmed by the software to provide the desired communications frequency. In addition, two additional 16 bit timers are left available to the applications programs to be used as event counters, real-time interrupts, etc.

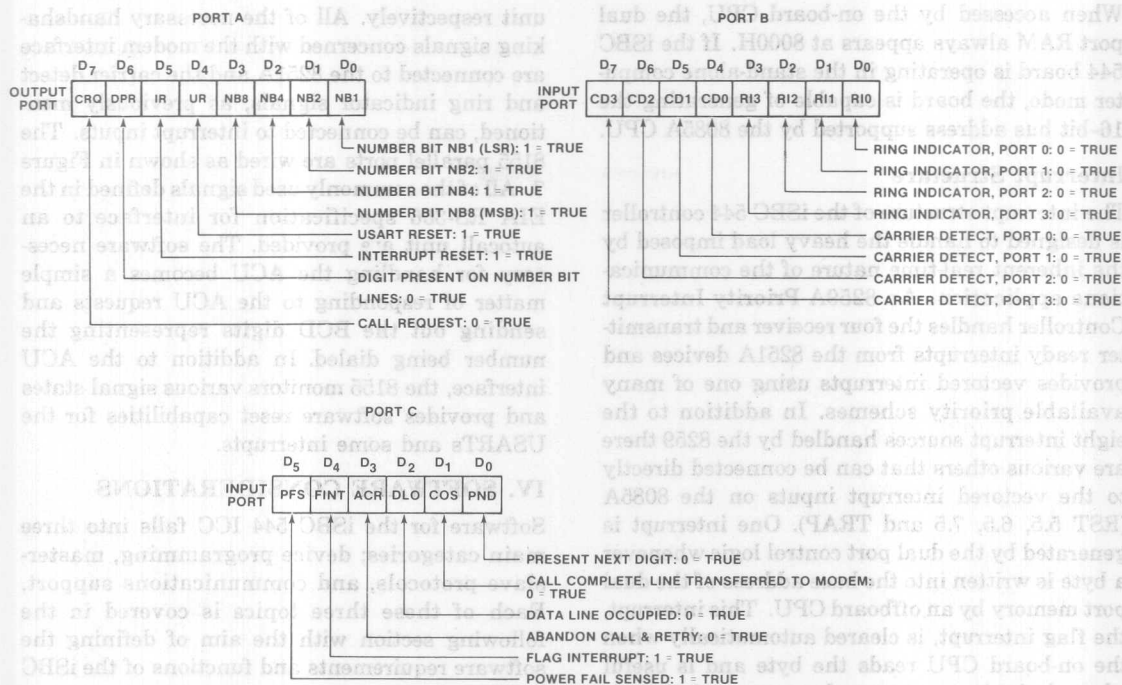


Figure 7. 8155 Pinout Definitions

The 8259A Priority Interrupt Controller is programmed to vector all interrupts through a jump table in memory. Also, the device provides software selectable priority schemes and an interrupt mask register for sophisticated interrupt management designs.

Last, but not least, the 8155 Programmable Peripheral Interface provides various software controlled input and output ports as discussed in previous sections. One specific point to remember is that the power on state of the 8155 clamps the reset signal to the USART's active and must be removed by programming the 8155 before communications can begin.

Master-Slave Protocols

If an application system is visualized at the highest level it appears to be a computer with various inputs and outputs as depicted in Figure 8a. If this computer is broken down into a master CPU and one or more intelligent slaves, great increases in efficiency and system throughput

can be realized by distributing the duties between the CPUs (Figure 8b). Once this split is performed, some well defined means of communication between master and slaves needs to be defined so that the processes that execute on the different machines can cooperate. This means of communication takes the form of a protocol followed by both master and slave.

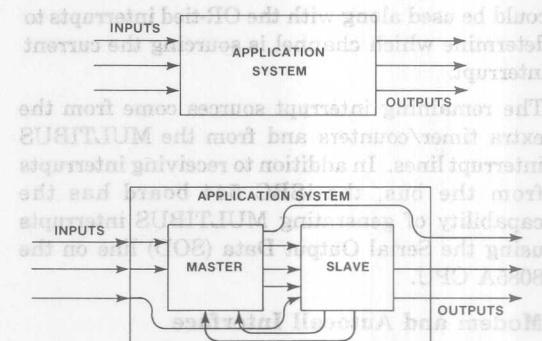


Figure 8a and 8b. System Software Block Diagrams

The intelligent slave architecture was designed to simplify the development of the necessary protocol. The shared memory space in the dual port RAM provides a large communications buffer area where data and commands can be transferred using normal memory transfers. Data structures of any needed complexity can be built in this memory area and accessed by both master and slave. The flag interrupt can be used to provide a unique synchronization signal from a master to a given slave. In addition, the MULTI-BUS interrupt lines can be used to provide extra signals in both directions. As we shall see in the system software section, these basic tools can be utilized to design a general purpose data transfer mechanism which isolates the applications processes from the worries of protocols and synchronization.

Communications Support

The previous software topics dealt mainly with the system overhead that must be handled by the communications processor. The larger and more important duty of the CPU is dealing with the application at hand—communications.

When configured as an intelligent slave to some master iSBC CPU board, the iSBC 544 board works to offload the master of communications related functions and at the same time is itself relieved of a major share of the system overhead and can be tuned to provide the highest possible throughput. With this combination, more complex applications can be tackled where the number of lines and the line frequencies are greatly increased. Multiple systems can be employed to provide a network facility with the iSBC 544 board now handling the network protocol in addition to its other duties. The architecture of the iSBC 544 controller is designed to simplify the user's software development process. The board can be programmed to handle many possible data transmission functions from simple line protocols to terminal control to link protocols and all the way up to network protocols.

In the stand-alone mode, the iSBC 544 board can assume total responsibility for the application. This can be done with on-board resources only or can include the support of offboard expansion like the iSBC 534 four channel serial controller. Appli-

cations of the stand-alone controller could include cluster controllers, peripherals managers, line concentrators or any other small system.

V. THROUGHPUT ANALYSIS

This section of the application note deals with studies that have been done to quantify the performance of the iSBC 544 board in both the stand-alone and intelligent slave modes. After describing the various test configurations and assumptions the data will be presented in graphical form and analyzed. The graphical data can be found in Appendix C.

Stand Alone Throughput

The first two tests were run to determine the absolute best case throughput of the iSBC 544 board configured as a stand-alone computer. Figure 9a shows the iSBC 544 controller continuously outputting data from four buffers to the four USARTs. Figure 9b shows essentially the same setup with eight channels, four on the iSBC 544 board and four on the iSBC 534 expansion card. In each configuration the 8251A was run in synchronous mode and the baud rate was incremented until the transmitter empty signal from the USARTs became active. Further increments of the baud rates would not have resulted in higher throughput since the CPU was already spending 100% of its available time responding to USART service requests.

The maximum rate for the first configuration (iSBC 544 board only) was 32,311 baud per channel. When the iSBC 534 expansion board was added a rate of 12,186 baud per channel was achieved. The drop in baud rate was due to the extra processing required by the offboard logic (eg. reading 8259 interrupt controller on the iSBC 534 board to determine which device is requesting service).

It should be noted that the serial throughput tests were run with almost no overhead and no actual processing of the data involved. The reader is expected to apply information on the amount of overhead expected in each individual application. For instance, if the application code for a given system is expected to utilize approximately 40% of the available CPU time and we wish to run four

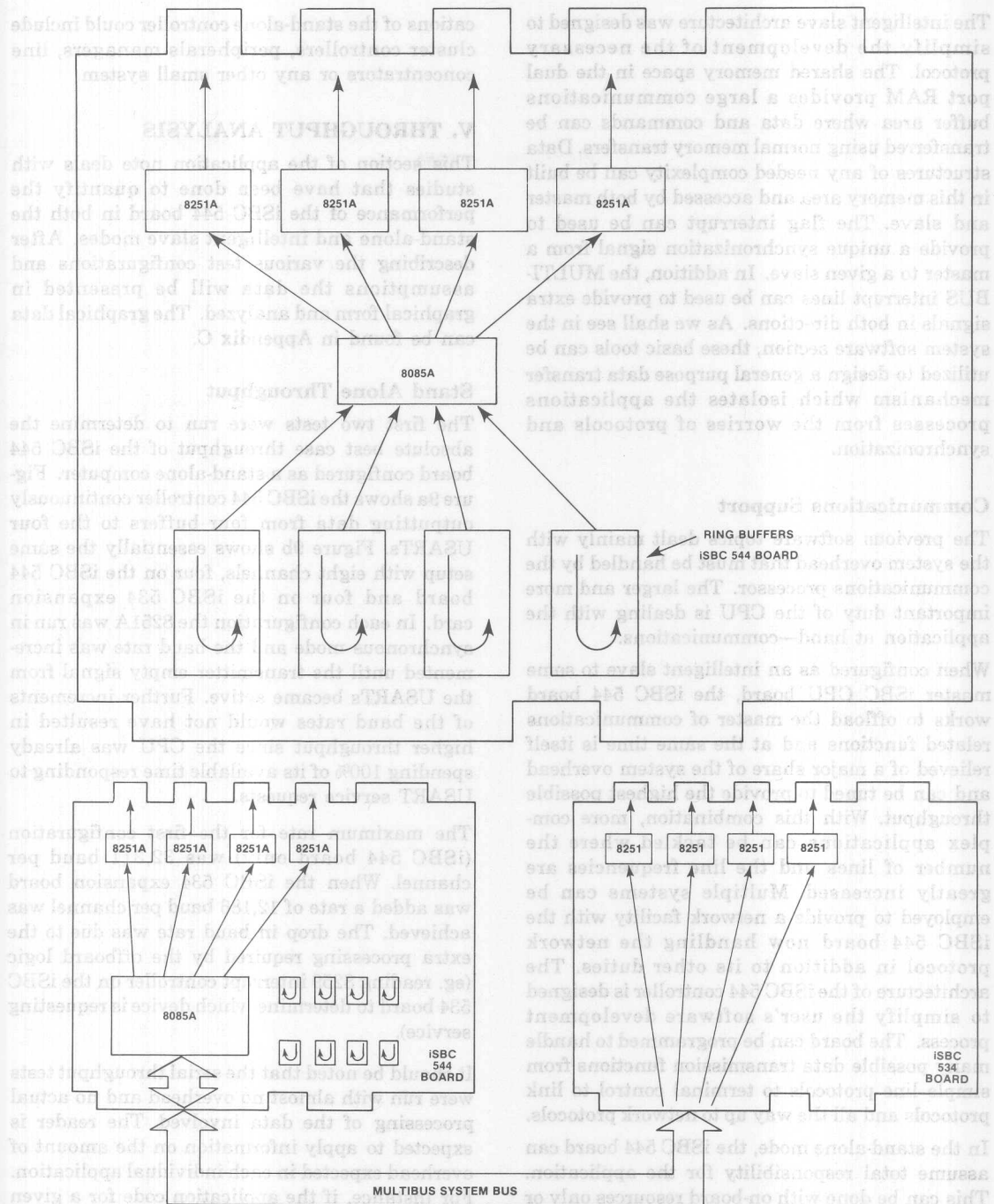


Figure 9a and 9b. Stand-Alone Throughput Configurations

full duplex channels in asynchronous mode the estimate of maximum baud rate would take the following form.

$$\begin{aligned}
 &32,331 \text{ baud per channel} - 40\% = 19,398.6 \text{ baud} \\
 &19,398.6 \text{ baud per channel synchronous} \times 10/8 \\
 &= 24,248.25 \text{ baud asynchronous} \\
 &24,248.25 \text{ baud per channel half duplex}/2 = \\
 &12,124.125 \text{ full duplex}
 \end{aligned}$$

Therefore, the maximum standard baud rate would be 9600 baud per channel in full duplex asynchronous mode.

Intelligent Slave Throughput

The remaining four configurations were set up to determine the effectiveness of the intelligent slave in the overall system. The general system configuration is illustrated in Figure 10. The boards surrounded by the box represent the systems under test. The disk controller and two iSBC 80/20 single board computers were active on the bus to simulate the normal bus traffic load in an application system. Various bus duty cycles were created using the computers and the disk controller to perform tasks that resulted in fixed bus utilization.

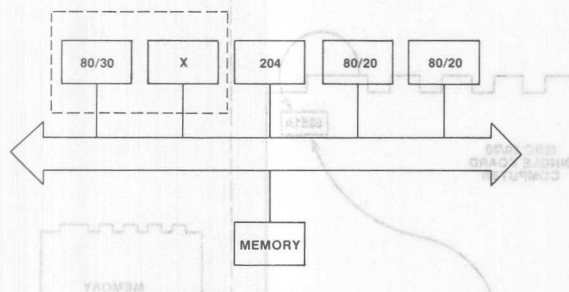


Figure 10. General System Configuration for Throughput Testing

In each configuration a single full duplex channel was set up with the input provided by another CPU. Only those functions dealing with system overhead were included and the data measured

reflected the amount of bus time, master CPU time and slave CPU time left available to applications oriented tasks. In each case this percentage of time available was measured as the baud rate was stepped up so that a graph could be constructed showing time available as a function of transmission speed.

CPU free time was measured using a counting program running in the background. After each USART interrupt the counter was started. As interrupts from other sources came in the counting was preempted and then resumed after servicing the interrupt. When the next USART interrupt occurred, the counter contents were examined and if the value was lower than the stored value the current value became the stored value. After ten minutes the stored value was retrieved and used as an indicator of the worst case time available between interrupts.

System bus utilization was measured using the circuit shown in Figure 11. The voltage measured by the digital voltmeter represented a time average of the voltage at the output of the flip-flop. A calibration chart was created using a pulse generator to simulate various duty cycles and then this chart was used to measure bus activity while the test was running.

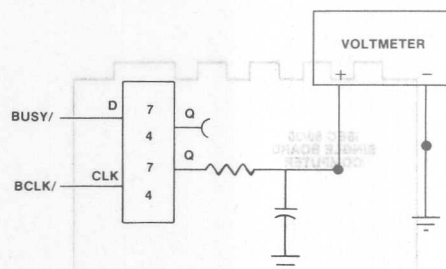


Figure 11. Bus Free Time Measurement Circuit

Configuration 1 is shown in Figure 12. This system uses a typical method of communications expansion with the iSBC 80/30 single board computer handling the lines directly via the serial I/O ports on the iSBC 534 I/O controller board.

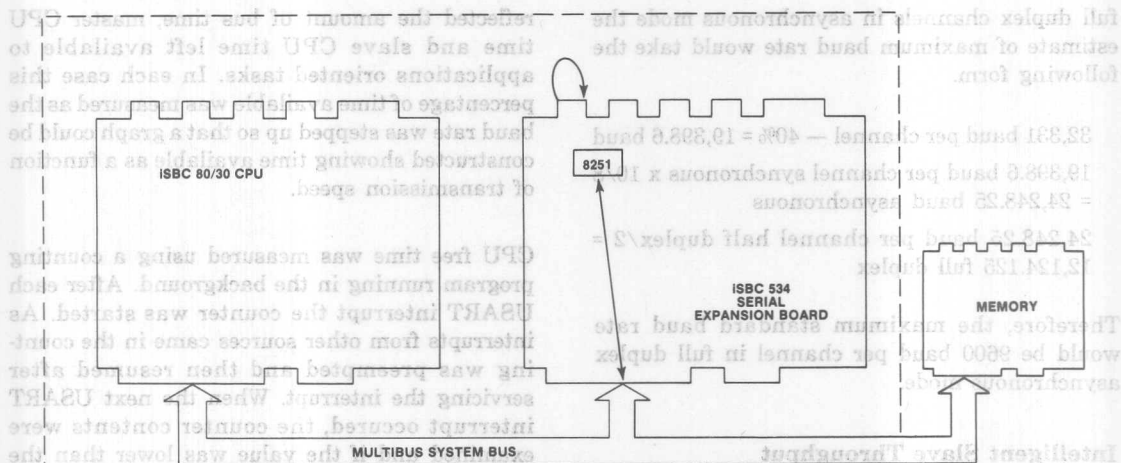


Figure 12. System Throughput Test, Configuration 1

The second configuration (Figure 13) illustrates the performance of the traditional DMA controller approach. If the communications controller had DMA logic instead of a dual port memory and transferred data directly into system memory the performance would be as observed in this test. In configuration 3 (Figure 14) the iSBC 544 board was used in the intelligent slave mode. This

configuration differs from the second in that memory transfers involved only local memory and bus access was not required on a per character basis.

The fourth and final configuration sought to identify the loading that additional intelligent slave controllers would impose on master CPU time and bus free time. Figure 15 shows the

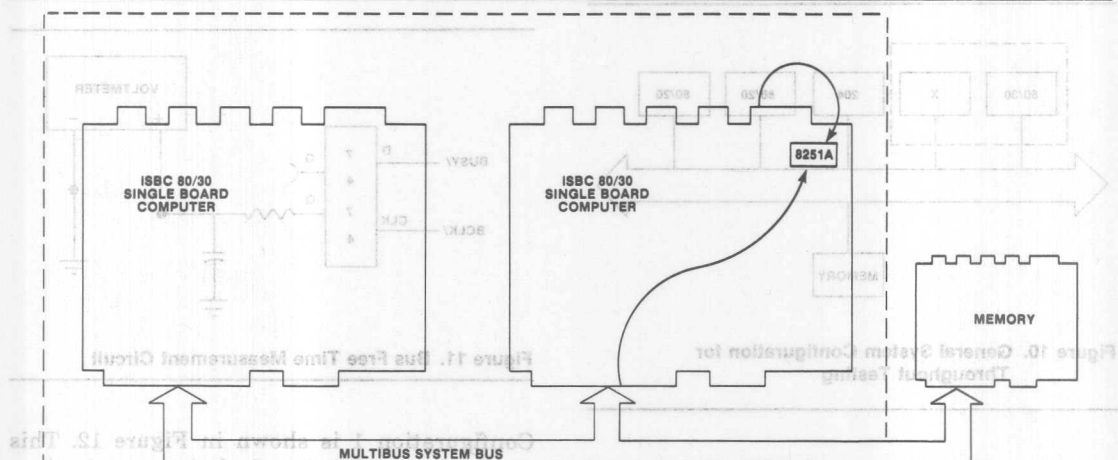


Figure 13. System Throughput Test, Configuration 2

The main objective of this study is to identify the factors of bus traffic on the performance of the various components of the system. The main observation to be made in this sequence is the drop in CPU free times and maximum bus rates that occur when the bus is heavily loaded. This is observed in the communications processor free time when the ISBC 544 expansion board or the DMA controller configuration is used. No effect is evident in the configuration with an ISBC 80/30 board.

The cause of this effect is the amount of bus access required by each configuration to move the characters from the ISART to or from the buffer. With an ISBC 80/30 board the master CPU receives an interrupt, polls the offboard 8255 interrupt controller, reads the character, stores it in system memory and sends an end of interrupt command to the offboard interrupt controller. When the ISBC 80/30 computer receives an interrupt all processing is performed on-board until a bus access is needed to move the 512 byte frame memory. In the case of the intelligent slave, all processing for a character is performed by the slave. The delay encountered becomes very fully utilized, the delays encountered in receiving bus access by the first two configurations become significant.

Figure 14. System Throughput Test, Configuration 3

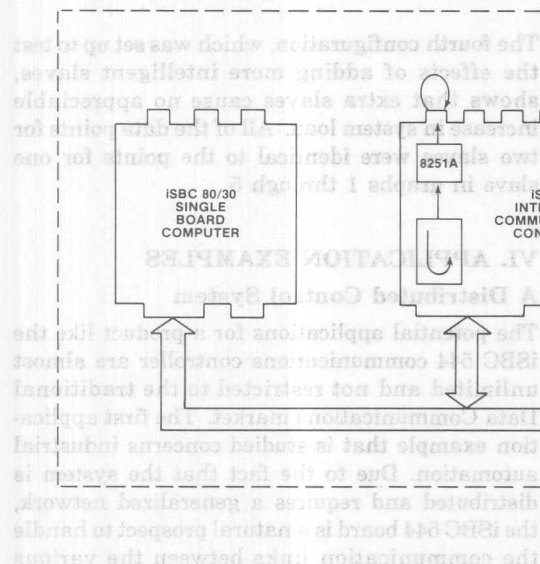


Figure 15. System Throughput Test, Configuration 4

The graphical presentation of the results is split into two sections. The first three graphs (Graph 1 through Graph 3) show the relationship between bus rates and the slave CPU free times. The slave CPU utilization. All of these results are based upon tests with 30% induced bus traffic. The two ISBC 80/30 computers and the ISBC 544 controller were active.

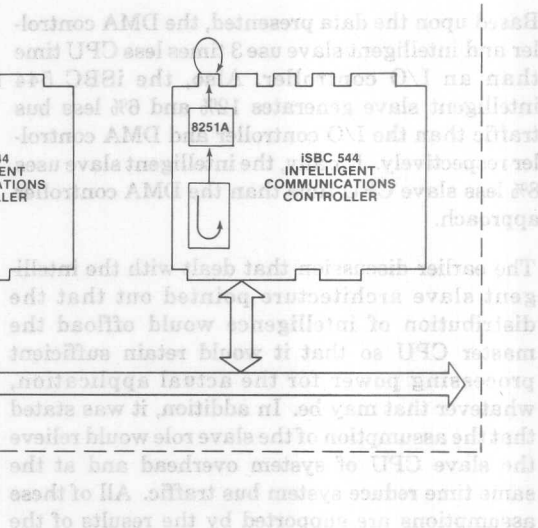
In Graph 4, time is graphed as a function of bus traffic. The processor in this case is the one actually involved with the data of a particular test. The ISBC 80/30 board in configuration 1, ISBC 80/30 board simulating DMA Controller in configuration 2, and ISBC 80/30 board in configuration 3.

Finally, Graph 5 illustrates the maximum attainable bus rate for each configuration as the bus traffic is increased.

All of the graphs identify the relative performance difference between the configurations. Absolute numbers are not presented due to the fact that the overhead imposed by the test software affects CPU free time measurements. Since the overhead applies equally to all configurations, the relative performance indications are valid.

Based upon the data presented, the DMA controller and intelligent slave use 3 times less CPU time than an ISBC 80/30 board. The ISBC 80/30 board intelligent slave requires 10% and 6% less bus traffic than the DMA controller. The intelligent slave requires 10% and 6% less bus traffic than the DMA controller. The intelligent slave requires 10% and 6% less bus traffic than the DMA controller.

The results indicate that both with the master and slave architectures, the distribution of intelligent would offload the master CPU so that it could retain sufficient processing power for the actual application. Whatever that may be. In addition, it was stated that the assumption of the slave would relieve the slave CPU of system overhead and at the same time reduce system bus traffic. All of these assumptions are supported by the results of the testing presented here.



configuration with two iSBC 544 boards executing identical programs.

The graphical presentation of the results is split into two sections. The first three graphs (Graph 1 through Graph 3) show the relationship between baud rates and the master CPU, system bus, and slave CPU utilization. All of these results are based upon tests with 30% induced bus traffic (i.e., the two iSBC 80/20 computers and the iSBC 204 disk controller were active.)

In graph 4, processor free time is graphed as a function of bus traffic. The processor in this case is the one actually involved with the data on a per character basis (i.e., iSBC 80/30 board in configuration 1, iSBC 80/30 board simulating DMA Controller in configuration 2, and iSBC 544 board in configuration 3).

Finally, graph 5 illustrates the maximum attainable baud rate for each configuration as the bus traffic is increased.

All of the graphs identify the relative performance difference between the configurations. Absolute numbers are not presented due to the fact that the overhead imposed by the test software affects the CPU time being measured. Since the overhead applies equally to all configurations, the relative performance indications are valid.

Based upon the data presented, the DMA controller and intelligent slave use 3 times less CPU time than an I/O controller. Also, the iSBC 544 intelligent slave generates 12% and 6% less bus traffic than the I/O controller and DMA controller respectively. Finally, the intelligent slave uses 8% less slave CPU time than the DMA controller approach.

The earlier discussion that dealt with the intelligent slave architecture pointed out that the distribution of intelligence would offload the master CPU so that it would retain sufficient processing power for the actual application, whatever that may be. In addition, it was stated that the assumption of the slave role would relieve the slave CPU of system overhead and at the same time reduce system bus traffic. All of these assumptions are supported by the results of the testing presented here.

The second set of graphs identify the effects of bus traffic on the performance of the various components of the system. The main observation to be made in this sequence is the drop in CPU free times and maximum baud rates that occurs when the bus gets busy. This effect is observable in the communications processor free time when the iSBC 534 expansion board or the DMA controller configuration is used. No effect is evident in the configuration with an iSBC 544 board.

The cause of this effect is the amount of bus access required by each configuration to move the characters from the USART to or from the buffer. With an iSBC 534 board the master CPU receives an interrupt, polls the offboard 8259 interrupt controller, reads in a character, stores it in system memory and sends an end of interrupt command to the offboard interrupt controller. When the iSBC 80/30 computer receives an interrupt all processing is performed onboard until a bus access is required to move the data byte from/to memory. In the case of the intelligent slave, all processing for a character is performed onboard. Thus, as the system bus becomes very fully utilized, the delays encountered in receiving bus access by the first two configurations become significant.

The fourth configuration, which was set up to test the effects of adding more intelligent slaves, shows that extra slaves cause no appreciable increase in system load. All of the data points for two slaves were identical to the points for one slave in graphs 1 through 5.

VI. APPLICATION EXAMPLES

A Distributed Control System

The potential applications for a product like the iSBC 544 communications controller are almost unlimited and not restricted to the traditional Data Communications market. The first application example that is studied concerns industrial automation. Due to the fact that the system is distributed and requires a generalized network, the iSBC 544 board is a natural prospect to handle the communication links between the various nodes in the system.

Design Requirements

The system to be designed is intended to provide the framework for a family of distributed control systems where the configurations and the objects to be controlled vary from system to system. Figure 16 shows the general picture of the system.

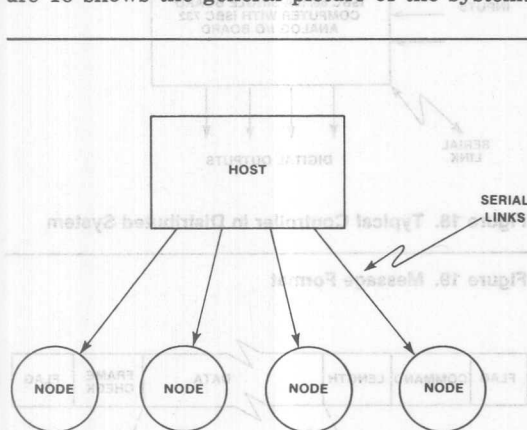


Figure 16. General Diagram of Distributed Control System

The host is responsible for providing supervisory control and a high-level human interface. The system can be expanded as shown in Figure 17 where the controllers attached to the host are replaced by intermediate nodes which contain controllers or other nodes. This process can be continued as far as is necessary to provide the needed number of controllers. Each controller in the diagram represents a localized closed loop control system that is tailored to the specific application.

The following system requirements need to be met by the computer network:

- The host CPU must have sufficient computational power to handle the human interface, mass storage management, supervisory control calculations and network control.
- The host CPU must not be overly burdened by low-level communications functions if it is to handle the other duties assigned to it.
- Node controllers must be capable of handling 8 medium speed lines and also modems and autocal units since the nodes or controllers attached may be remote.

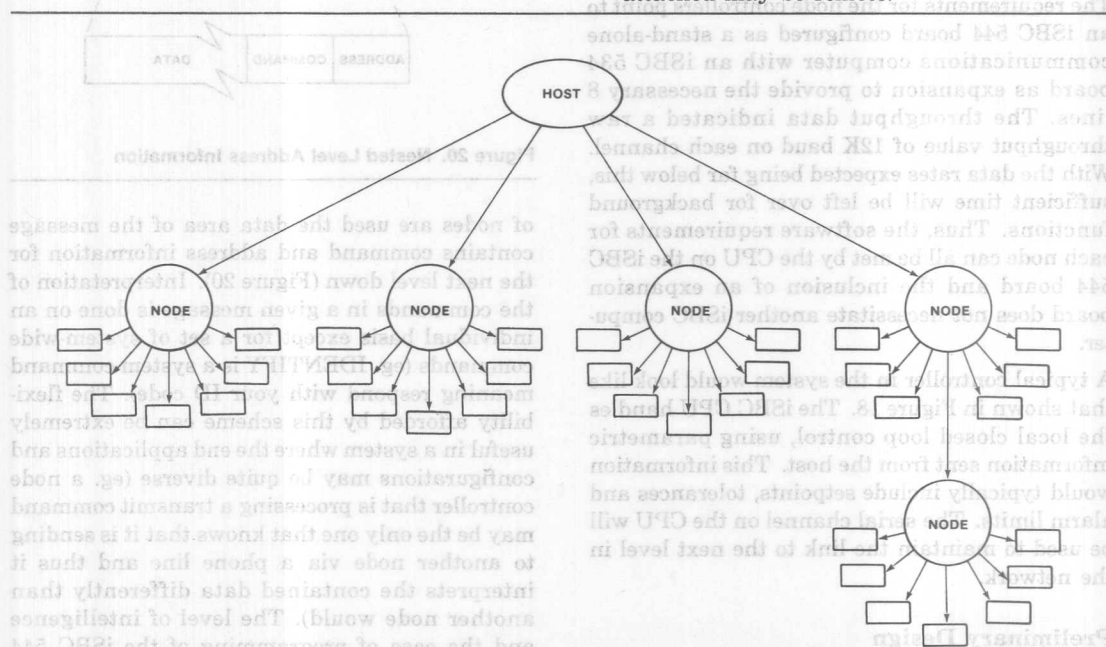


Figure 17. Expanded Diagram of Distributed Control System

- The message transmission format must be independent of the configuration and end application. The nodes in the network must be capable of passing through messages with and without interpreting the contained data.
- The system must be capable of auto-configuration (since the network configuration is tailored to the specific application, the host must be able to automatically determine the setup at power on).
- Each node controller is responsible for verifying the integrity of the nodes attached.

System Configuration

Based upon the design criteria and the benchmark information the chosen configuration uses an iSBC 86/12 Single Board Computer as the host with an iSBC 544 intelligent slave handling the communications load for the CPU. The USART on the CPU board will talk to the local terminal and an iSBC 206 Hard Disk Controller will be used to provide up to 40 Megabytes of mass storage capacity.

The requirements for the node controllers point to an iSBC 544 board configured as a stand-alone communications computer with an iSBC 534 board as expansion to provide the necessary 8 lines. The throughput data indicated a raw throughput value of 12K baud on each channel. With the data rates expected being far below this, sufficient time will be left over for background functions. Thus, the software requirements for each node can all be met by the CPU on the iSBC 544 board and the inclusion of an expansion board does not necessitate another iSBC computer.

A typical controller in the system would look like that shown in Figure 18. The iSBC CPU handles the local closed loop control, using parametric information sent from the host. This information would typically include setpoints, tolerances and alarm limits. The serial channel on the CPU will be used to maintain the link to the next level in the network.

Preliminary Design

The message format that the system uses is shown in Figure 19. When multiple nested levels

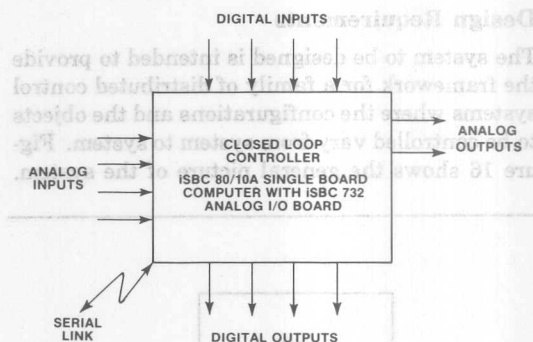


Figure 18. Typical Controller in Distributed System

Figure 19. Message Format

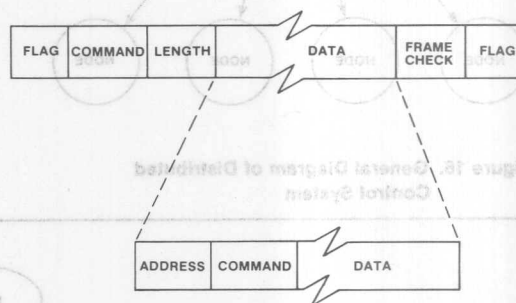


Figure 20. Nested Level Address Information

of nodes are used the data area of the message contains command and address information for the next level down (Figure 20). Interpretation of the commands in a given message is done on an individual basis except for a set of system-wide commands (eg. IDENTIFY is a system command meaning respond with your ID code). The flexibility afforded by this scheme can be extremely useful in a system where the end applications and configurations may be quite diverse (eg. a node controller that is processing a transmit command may be the only one that knows that it is sending to another node via a phone line and thus it interprets the contained data differently than another node would). The level of intelligence and the ease of programming of the iSBC 544 board make this generalized transmission scheme possible.

The simplest means of auto-configuration requires each controller in the system to send an identity message to the nearest node. This node would know the logical address of the controller that sent the message and would attach this address to the message and retransmit it to the next level as illustrated in Figure 21. This process would be repeated until the host is reached and would contain, at this point, all necessary address information to reach the given controller.

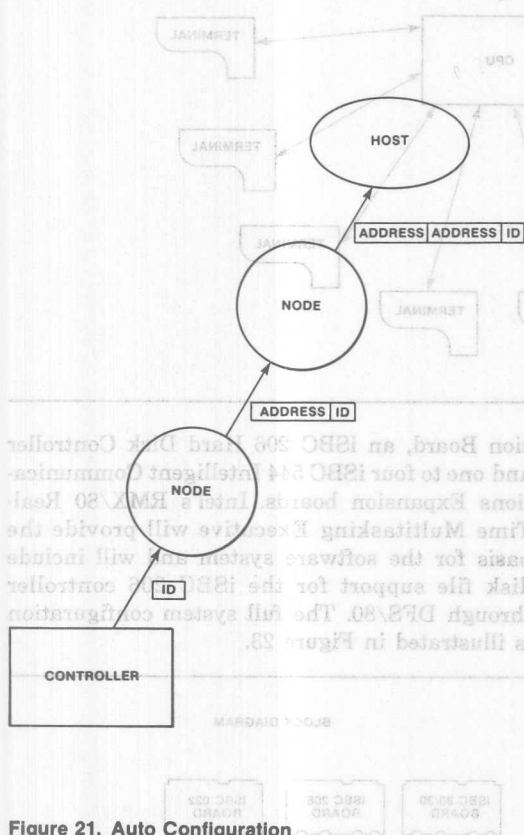


Figure 21. Auto Configuration

The human interface on the host would provide a mapping mechanism to attach meaningful symbolic names to the various nodes in the system. This labeling, along with the application specific control algorithms, make it possible to say something like "lower the temperature on the third floor to 68°F". The host breaks this information down into setpoints and tolerances,

uses the map to determine the path to the node(s) responsible for the third floor and transmits the information through the network.

Each node controller in the system has the added responsibility of verifying the integrity of all the nodes attached to it. This duty can be handled by periodic background commands issued from the host and propagated through the network. Each node is responsible for passing the command along and also polling the nodes attached to it and reporting back any error conditions.

Summary

Through the use of a powerful 16-bit iSBC Single Board Computer, various low-cost 8-bit iSBC CPUs and the iSBC 544 communications controller, a flexible and extensible distributed control system is easy to design. The dual nature of the iSBC 544 board provides both an intelligent front end to the host computer and a high-speed stand-alone nodal concentrator. The ability to individually customize the software on each controller provides for an easily expandable system design.

Terminal Cluster Controller

The second application example concerns itself with a terminal cluster controller. The system shown in Figure 22 uses a number of "dumb" terminals and makes them appear "intelligent" via a local microcomputer system. The local microcomputer interfaces with the operator and accesses a local data base to provide an inquiry and data entry service. When necessary, the local microcomputer is capable of calling the host via an autocall unit and exchanging information and updates to the data base.

Design Criteria

The terminal cluster controller must meet the following criteria:

- Support must be provided for from four to sixteen operator terminals all running at rates up to 2400 baud.
- Line editing on input must be provided (delete characters, delete lines and pause output).

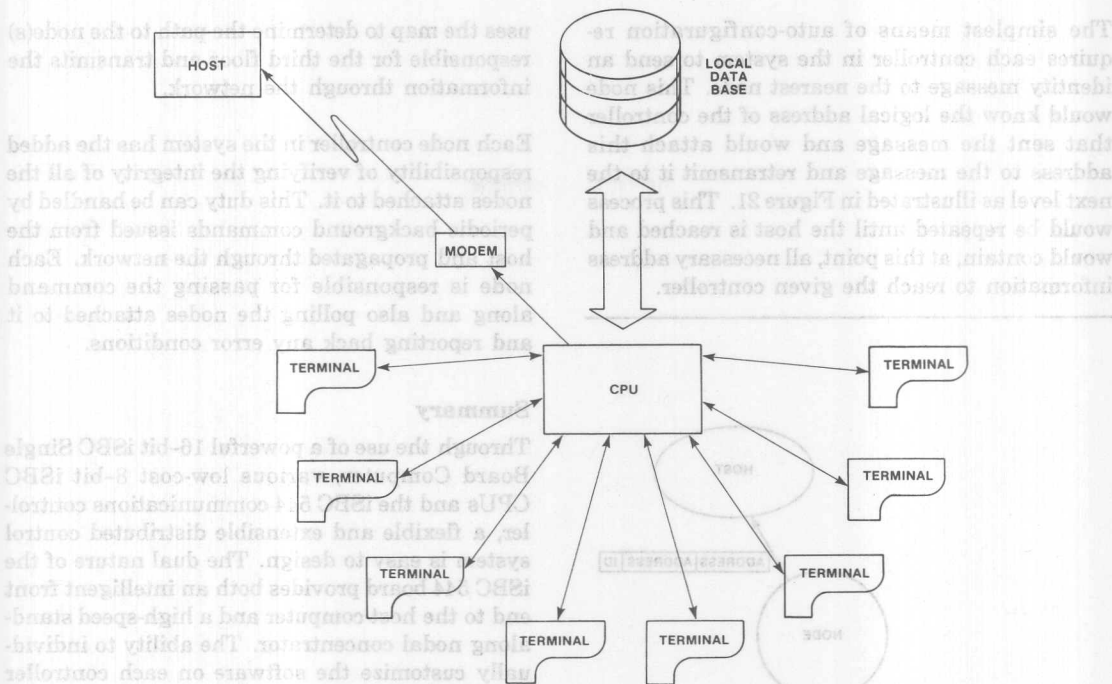


Figure 22. Terminal Cluster

- Support for the terminals must be configurable in that certain stations may require different screen formats.
- Support for an optional hard copy device must be allowed for.
- A considerable amount of CPU free time must be available after the basic terminal facilities are included. This is due to the fact that the data base management software to be written to run on the master single board computer will be extensive.
- Type ahead would be a desired feature since the processing on the master CPU after a line of input has been transmitted may cause a delay in responding and we would like to have the ability to continue entering input while waiting for the response.

System Configuration

The specific iSBC products needed to implement the system described are the iSBC 80/30 Single Board Computer with an iSBC 032 RAM Expan-

sion Board, an iSBC 206 Hard Disk Controller and one to four iSBC 544 Intelligent Communications Expansion boards. Intel's RMX/80 Real-Time Multitasking Executive will provide the basis for the software system and will include disk file support for the iSBC 206 controller through DFS/80. The full system configuration is illustrated in Figure 23.

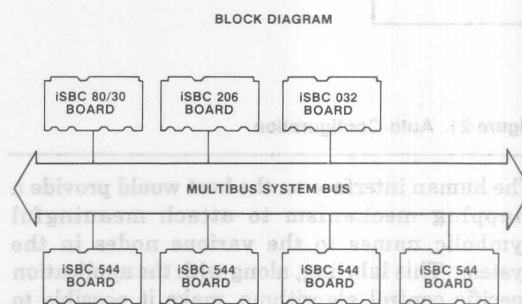


Figure 23. Terminal Cluster Controller System Configuration

Preliminary Design

The first design decision to be made involves the distribution of system functions. Due to the requirements for line-editing and type-ahead the software for processing characters input from the terminal keyboards will be somewhat lengthy. The standard terminal output handler will be very small but provisions for special screen format controls and/or hard copy devices must be allowed for. All of these requirements lead to the use of the iSBC 544 controller for all terminal functions. If the master CPU were burdened with all of these duties it would be unable to adequately perform its data base management functions. The fast CPU and 8K PROM capacity of the iSBC 544 board will be more than adequate for the task at hand.

The throughput tests indicate that the loading imposed by expanding the number of terminals (and therefore the number of iSBC 544 boards) will not adversely affect the performance of the rest of the system. Master CPU free time and bus traffic data for two intelligent slaves in the system were identical to the numbers for one slave. Thus, since the iSBC 80/30 single board computer and the MULTIBUS system bus can handle one iSBC 544 controller they can also handle the maximum of four controllers that may be required by this application. The only observable effect will be caused by the load the extra operators impose on the data base software itself.

The software needed for the iSBC 544 board is now defined and divided into three major pieces; a terminal input handler, a terminal output handler and system software to support the handlers. Since the input and output handlers are invoked via USART interrupts, all that need be done is to write a single routine for each handler and have it talk to all of the devices on the board. This can be accomplished by vectoring the proper interrupts to the entry point of the routine and then polling the 8259A interrupt controller to determine which device needs servicing.

The standard terminal input handler needs to read in the available character from the USART,

check it to see if it is a special command character and, if not, store it into a buffer. If a command character is encountered, the handler will respond by performing the appropriate operation.

The standard terminal output handler simply takes characters out of a buffer upon interrupt from the transmitter and sends them to the appropriate USART. If a different output handler needs to be substituted for a special terminal or a hard copy device, a new routine can be included by modifying the interrupt vector address in the 8259A jump table.

Since the RMX/80 Real-Time Multitasking Executive is being utilized on the master CPU it is desirable to create an RMX/80 handler for the iSBC 544 boards that accepts and processes normal terminal handler request messages. In this manner, application tasks that formerly communicated with the on-board USART via the RMX/80 Terminal Handler can be made to talk to one of the devices on the iSBC 544 board by simply changing the address of an exchange. The following paragraphs, as well as paragraphs in the section on system software, assume a knowledge of the RMX/80 Real-Time Executive. This knowledge is not necessary to use the information contained in this application note. Interested readers are referred to the RMX/80 references listed in the front-piece.

Since this application can have from one to four iSBC 544 boards the RMX/80 driver will need to be configurable. A set of tasks and exchanges will be created for each terminal in the system. One task and exchange pair will accept and process terminal input request messages while another pair will process terminal output requests.

The remaining piece of software that is needed by this system will provide the means for getting commands and data between the master and intelligent slave. Since this is a common need in any system utilizing an intelligent slave we will develop a general purpose scheme that can be used by any application. In this manner, a routine such as the terminal input handler can be written without any concern for how it will get the data it is inputting to the master CPU; all it need do is call upon a standard routine to "transmit"

the data. With these thoughts in mind, the following section discusses the system software developed for master-intelligent slave communication. After the discussion of the system software we will revisit the software for the second application as an example of the use of the data transfer routines.

VII. SYSTEM SOFTWARE

In the earlier discussion of master-slave protocols, the notion was presented of developing a general purpose data transfer scheme which would enable the applications routines on both the slave and master to operate without concerning themselves with protocols and synchronization. This scheme can be implemented by designing a set of primitive routines to handle the data transfer activities. Thus, Figure 8b is expanded as shown in Figure 24 and the applications processes now call upon the primitives to handle the communications between the master and the slave.

Data Transfer Primitives

The basic mechanism used by this implementation of the primitives is a wraparound queue as shown in Figure 25. Each 8251A device has associated with it, in dual port memory, an input and an output queue each of which have a *give*

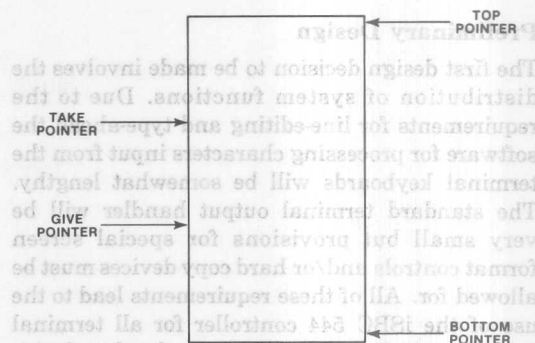


Figure 25. Wrap-around Queue Used by Data Transfer Primitives

and a *take* pointer. The *give* pointer contains the address of the next location in the queue that is available for filling with data. The *take* pointer contains the address of the next byte in an output queue that has been filled and is available. A queue is empty when the *give* and *take* pointers are equal and it is full when the act of incrementing the *give* pointer would make it equal to the *take* pointer. A *wrap* function is defined to increment a pointer such that an increment past the bottom of the queue "wraps" the pointer around to the top of the queue.

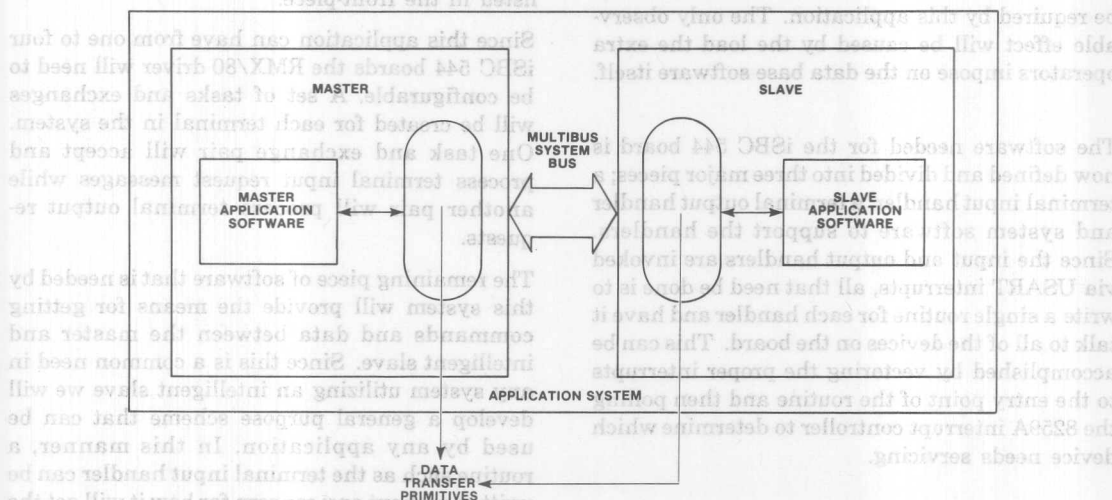


Figure 24. System Software Diagram with Data Transfer Primitives

The primitives all make use of a queue information block located at the base address of the slave's dual port memory (Figure 26). All pointer information is base relative to accommodate the needs of the two CPUs who have different memory maps. The two flag bytes carry information for master-slave and slave-master synchronization signals.

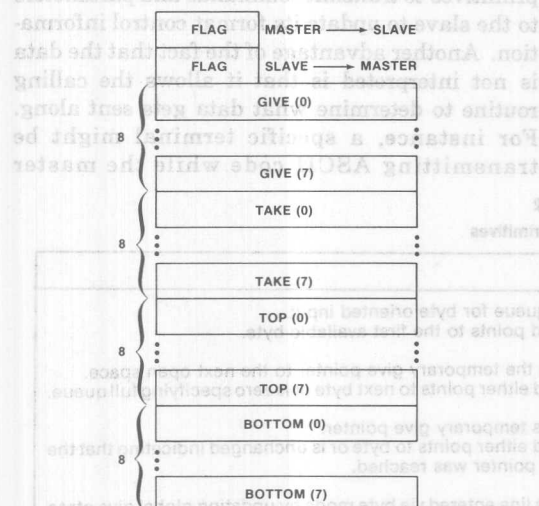


Figure 26. Queue Information Block

The set of primitives provides two distinct methods of information transfer, line oriented and byte oriented. The line oriented primitives are listed in Table 1. Both *get\$line* and *send\$line* transfer information between the queues and buffers provided by the caller. The disadvantage of this scheme is the number of memory moves needed to transfer information. The advantages of the line oriented method are the relative efficiencies and the simplicity of the interface from the calling routine.

The byte oriented primitives (Table 2) allow the calling routine to transfer data directly into and out of the queues. An example of the sequence for putting a character into a queue is illustrated in Figure 27. The routine servicing the receiver ready interrupt calls *next\$space* to get a pointer to the next available slot in the queue and then uses this pointer to transfer the data byte directly into the queue. The *new\$line*, *xmit*, *open\$line* and *receive* primitives are necessary since the global *give* and *take* pointers cannot be modified until all manipulations on the affected section of the queue are complete. If the pointers were modified continuously the routine gathering the data from the other side may see invalid data.

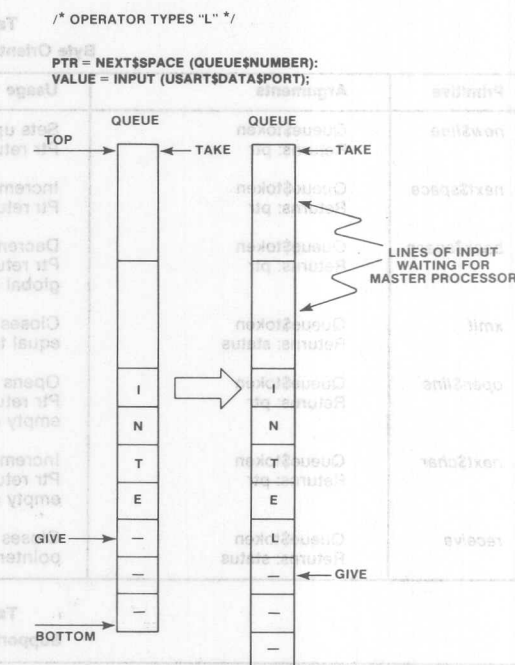


Figure 27. Sequence for Putting Data Into Queue

Table 1

Line Oriented Primitives

Primitive	Arguments	Usage
<i>send\$line</i>	Queue\$token, buf\$ptr, count Returns: overflow	Inserts count characters into queue from buffer If insufficient room available, overflow indicates how many would not fit
<i>get\$line</i>	Queue\$token, buf\$ptr, count Returns: Actual	Retrieves count characters from queue and puts them in buffer Actual indicates how many were actually moved

The remaining primitive routines deal with the general purpose needs of the application software with regard to interrupts, initialization and status checking. A full list of these support routines is contained in Table 3.

There are many features of this implementation and a few of them should be pointed out at this time. By defining a general purpose set of primitive routines to handle the data transfer, the actual means by which the bytes are transferred between slave and master is not visible to the calling routine. If the actual mechanism used needs to be altered the change will not affect the application software as long as the same external interface is maintained.

Another important feature of the primitive routines is the fact that they do not interpret the bytes that are sent to them. Due to this fact, the applications routines are free to send commands and parameters interspersed with the actual data. As an example, the terminal driver on an iSBC 544 board might perform format control based upon table information. The master applications software could use the data transfer primitives to transmit commands and parameters to the slave to update its format control information. Another advantage of the fact that the data is not interpreted is that it allows the calling routine to determine what data gets sent along. For instance, a specific terminal might be transmitting ASCII code while the master

Table 2
Byte Oriented Primitives

Primitive	Arguments	Usage
<i>new\$line</i>	Queue\$token Returns: ptr	Sets up a queue for byte oriented input. Ptr returned points to the first available byte.
<i>next\$space</i>	Queue\$token Returns: ptr	Increments the temporary give pointer to the next open space. Ptr returned either points to next byte or is zero specifying full queue.
<i>back\$space</i>	Queue\$token Returns: ptr	Decrements temporary give pointer. Ptr returned either points to byte or is unchanged indicating that the global give pointer was reached.
<i>xmit</i>	Queue\$token Returns: status	Closes off a line entered via byte mode by updating global give ptr to equal temporary give ptr. Status is either "normal" or "null".
<i>open\$line</i>	Queue\$token Returns: ptr	Opens up a line for byte oriented output. Ptr returned either points to the next byte or is zero indicating an empty queue.
<i>next\$char</i>	Queue\$token Returns: ptr	Increments temporary take pointer. Ptr returned either points to next byte or is zero indicating an empty queue.
<i>receive</i>	Queue\$token Returns: status	Closes off a line retrieved in byte mode by updating global take pointer to equal temporary ptr. Status is either "normal" or "null".

Table 3
Support Routines

Primitive	Arguments	Usage
<i>get\$status</i>	Queue\$token Returns: status	Returns status of queue. Possible values are "normal", "empty", "full" and "null".
<i>set\$interrupt</i>	Queue\$token, type Returns: status	Generates a slave → master or master → slave interrupt. Type code 0 is illegal and codes 8H → 0FH are reserved for use by the primitives.
<i>set\$handler</i>	Queue\$token, handler\$adr Returns: status	Inserts address into vector table used for handling interrupts described above.
<i>s\$init</i>	none	Called from slave software to initialize.
<i>m\$init</i>	none	Called from master software to initialize.

software is expecting EBCDIC. The routine on the slave can very easily perform the necessary code conversion before stuffing the data into a queue.

Sample Slave Software

Given the existence of the primitive routines the applications routines on the slave and master can deal with the specific duties of each device. The following paragraphs revisit the code from application example 2, first for the slave and then for the master. Full code listings for these programs can be found in Appendix D.

The flowchart for the terminal input handler resident on the iSBC 544 board is shown in Figure 28. Support is provided for deleting characters (Rubout), deleting lines (control-X), pausing and resuming output (control-S and control-Q) and terminating lines (escape and carriage return). The sections of code reproduced below use this terminal input handler to present an example of the use of the data transfer primitives to enter and edit a line of input from a terminal. The byte variable *value* is based on the address variable *value\$ptr* which is assigned by calls to the primitives. The routine *var\$inp* inputs and returns a data byte from an I/O port specified by a calling parameter. This is necessary since the particular USART to be serviced is determined by reading the 8259A in-service register.

```

/* case 1: rubout; delete char */
do;
  new$ptr=back$space(token);
  if new$ptr=length$ptr then
    dummy=echo(token+1,.(bell),1);
  else
    do;
      dummy=echo(token+1,.(BS,SP,BS),3);
      ptr=new$ptr;
      count=count-1;
    end;
end;

```

Following this, the byte input is checked to see if it is a control character and if so a block within a DO CASE statement is executed. As an example of one of these blocks, if the character input was a RUBOUT the code sequence below is executed. The *back\$space* primitive is called and a temporary pointer is returned to a location in the queue. A check is made to determine if the line was empty and, if so, a bell is echoed to signal the operator. If the pointer returned did not indicate

an invalid RUBOUT the real pointer is assigned the value of the temporary pointer and a backspace, space, backspace is echoed to delete the previous character on the screen. Lastly, the character count for the current line is decremented.

```

VALUE$PTR=NEXT$SPACE (QUEUE$NUMBER);
VALUE=VAR$INP (USART$DATA$PORT (NUM));

```

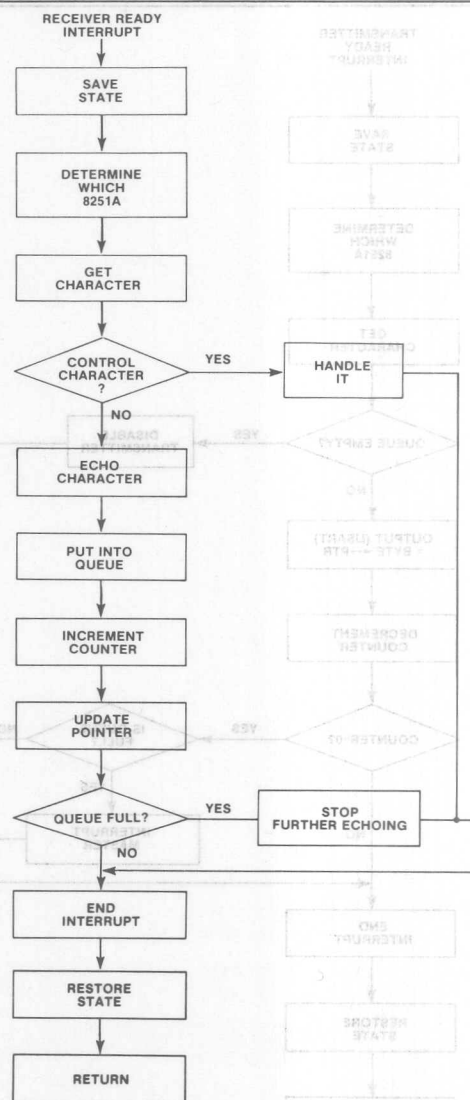


Figure 28. Flow Chart for Terminal Input Handler

In order to facilitate retrieval of the proper amount of information on the master side, the first byte of each message is defined to contain the number of characters in the message. Thus, when the master routine needs a line of input he uses the first byte as a count to retrieve the full line. The requirement for type-ahead is met by this mechanism since the number of lines in the

queue at a given time is limited only by the length of the queue. When a full line of input is finished, the terminal input handler generates a slave to master interrupt to signal the master routine who may be waiting for this event.

The flowchart for a minimal terminal output handler is shown in Figure 29. Upon receipt of a

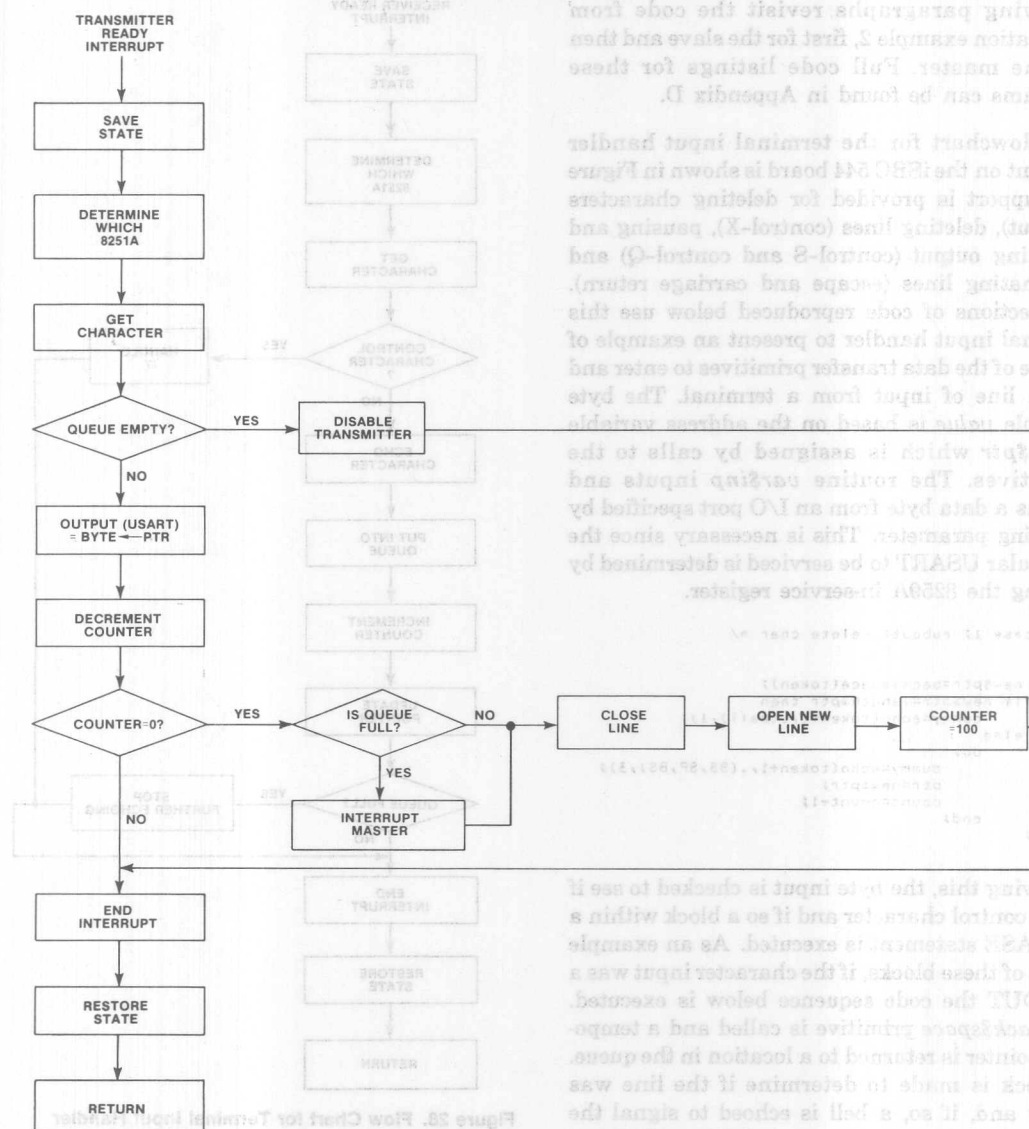


Figure 29. Flow Chart for Terminal Output Handler

transmitter ready interrupt the output handler requests a character from the appropriate queue. If one is available it is output to the USART. If the queue is empty, the transmitter is disabled. Whenever the master routine sends a line into the queue it will generate an interrupt to signal the slave handler and the transmitter will be reenabled. A line is opened via a call to *open\$line* and it is kept open until 100 characters have been retrieved via calls to *next\$byte*. At this time the line is closed by a call to *receive* making the space available to be reused. After this, a new call to *open\$line* starts the process over again. If the call to *get\$status* shows that the queue was full prior to the call to receive, an interrupt is sent to the master to reawaken any routine that may have been waiting for room in the queue to become available.

Sample Master Software

The RMX/80 handler for the master single board computer that will communicate with the software on the iSBC 544 board is diagrammed in Figure 30. In addition, the RMX/80 message used to convey information to the handler is shown on the right. The full software diagram is illustrated in Figure 31.

The input driver tasks execute a reentrant routine that services a request exchange that is specified in an initialization block that is unique to each of the input tasks. The necessary information is extracted from the request message and the *get\$line* primitive is called upon to get a line of input from the queue. If the call to *get\$line* for the length byte is unsuccessful the input task waits at

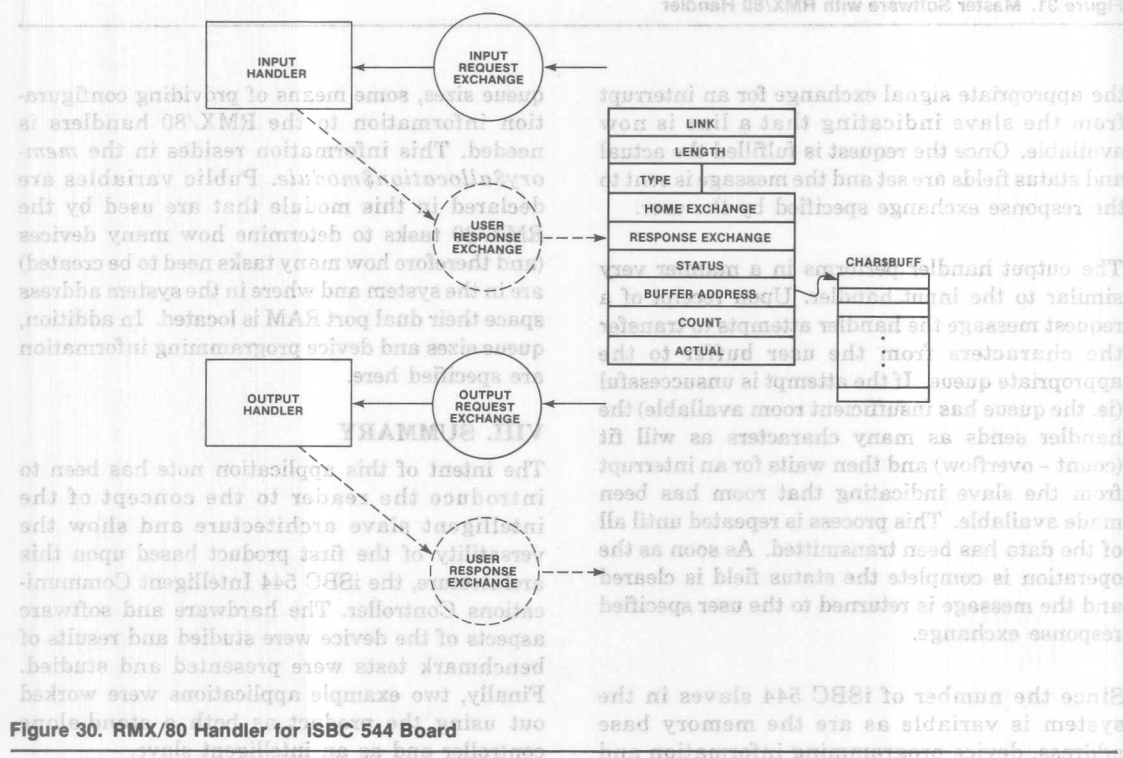


Figure 30. RMX/80 Handler for iSBC 544 Board

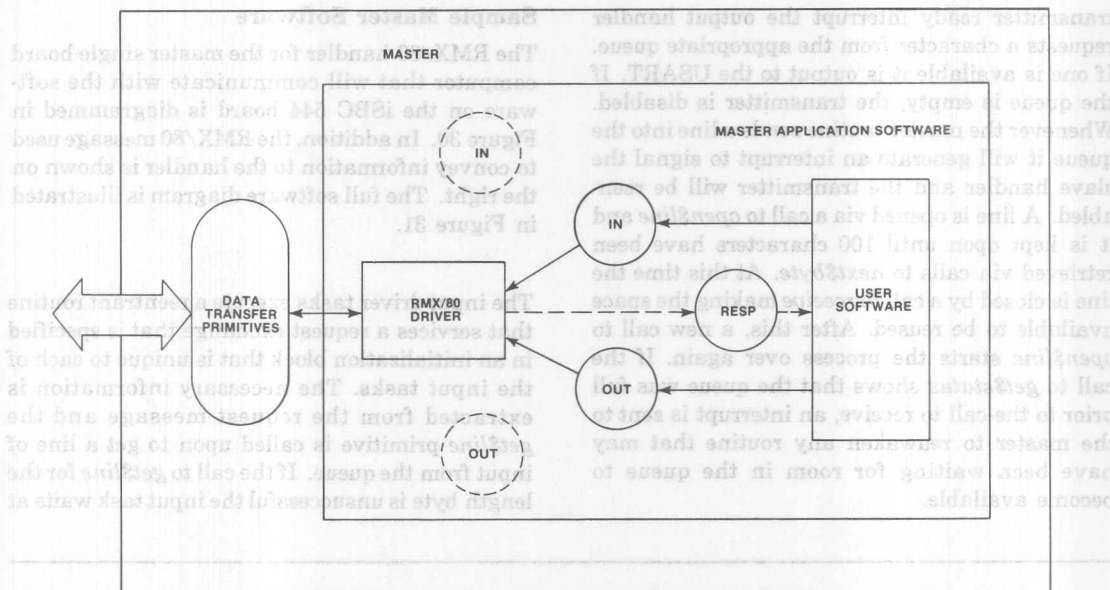


Figure 31. Master Software with RMX/80 Handler

the appropriate signal exchange for an interrupt from the slave indicating that a line is now available. Once the request is fulfilled the actual and status fields are set and the message is sent to the response exchange specified by the user.

The output handler performs in a manner very similar to the input handler. Upon receipt of a request message the handler attempts to transfer the characters from the user buffer to the appropriate queue. If the attempt is unsuccessful (ie. the queue has insufficient room available) the handler sends as many characters as will fit (count - overflow) and then waits for an interrupt from the slave indicating that room has been made available. This process is repeated until all of the data has been transmitted. As soon as the operation is complete the status field is cleared and the message is returned to the user specified response exchange.

Since the number of iSBC 544 slaves in the system is variable as are the memory base address, device programming information and

queue sizes, some means of providing configuration information to the RMX/80 handlers is needed. This information resides in the *memory\$allocation\$module*. Public variables are declared in this module that are used by the RMX/80 tasks to determine how many devices (and therefore how many tasks need to be created) are in the system and where in the system address space their dual port RAM is located. In addition, queue sizes and device programming information are specified here.

VIII. SUMMARY

The intent of this application note has been to introduce the reader to the concept of the intelligent slave architecture and show the versatility of the first product based upon this architecture, the iSBC 544 Intelligent Communications Controller. The hardware and software aspects of the device were studied and results of benchmark tests were presented and studied. Finally, two example applications were worked out using the product as both a stand-alone controller and as an intelligent slave.

The bottom line is that the iSBC 544 controller, due to the advanced architecture around which it is designed, can be the means to the end for any application that requires communication. The dual nature of the controller provides the full power of a single board computer to the small application while the large system can make use

of the fully programmable intelligent slave to free the CPU for complicated processing duties.

I would like to extend my gratitude to Dave Jurasek for the work on the throughput testing and to Jack Tyler Inman for aid in the design of the system software.

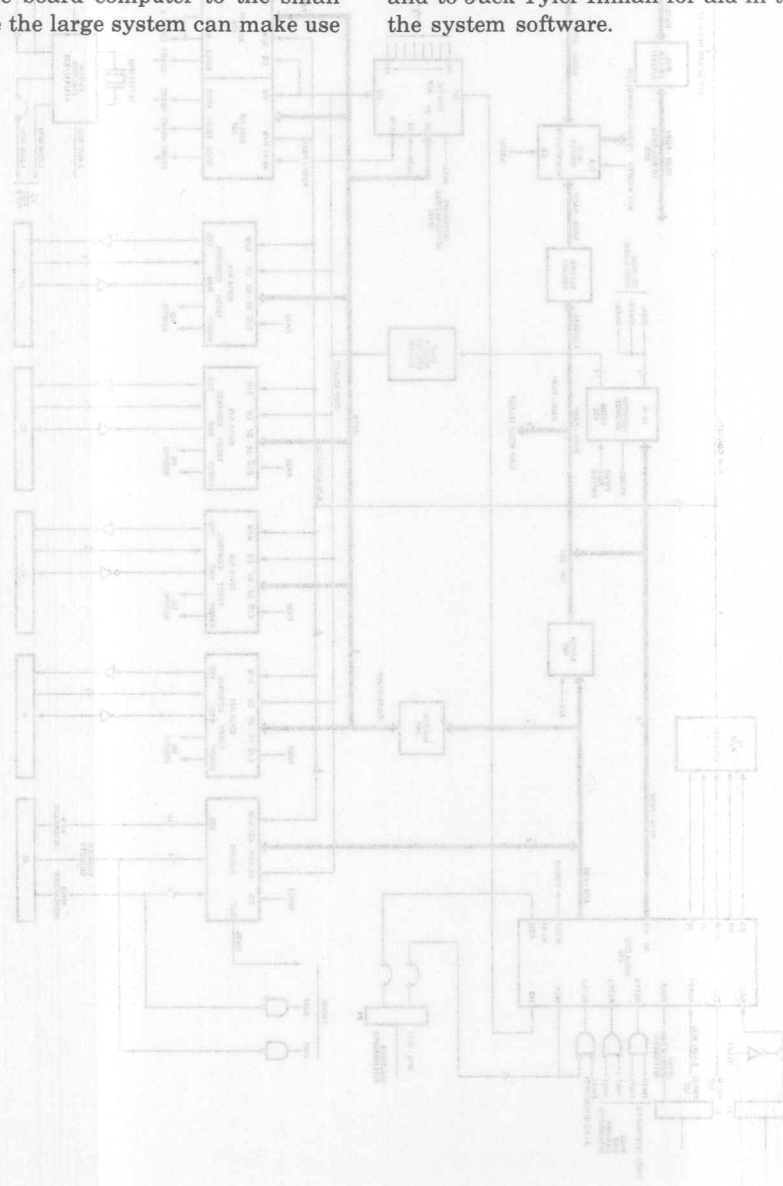


Figure A-1. iSBC 544 Input/Output and Interrupt Block Diagram

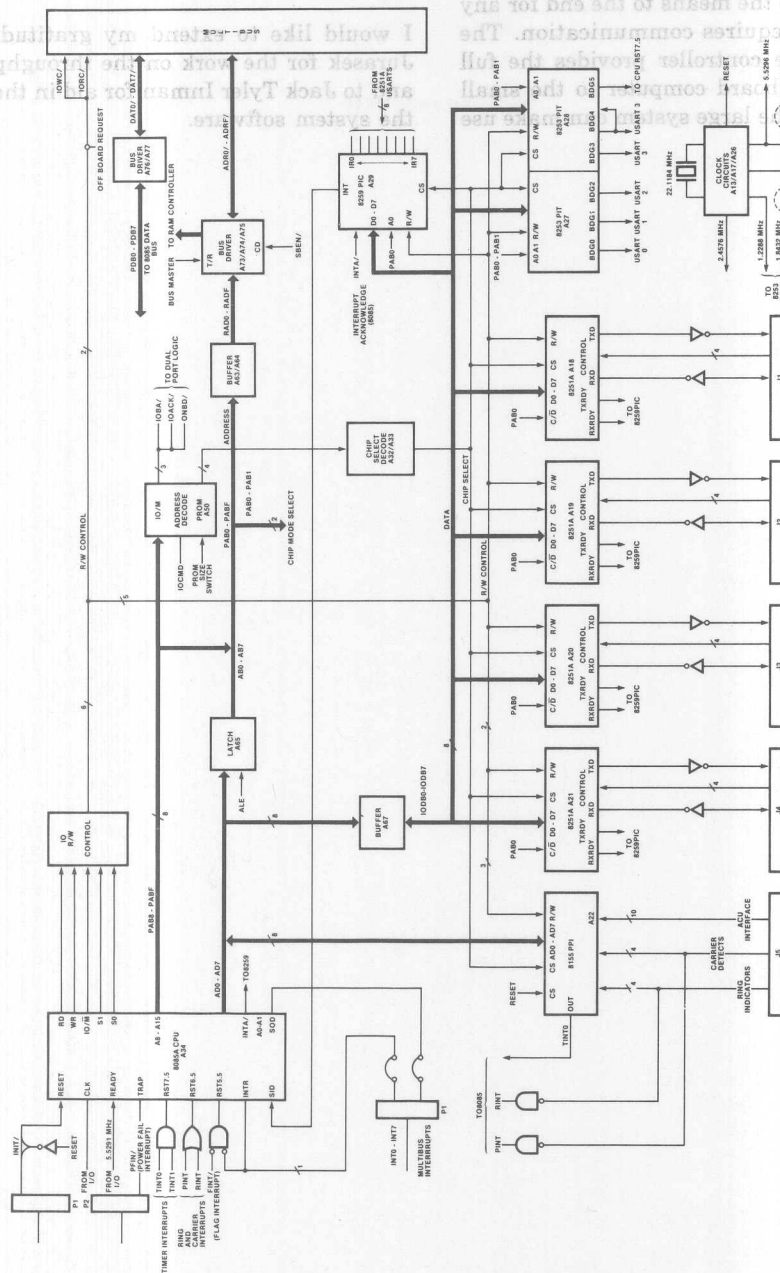


Figure A-1. ISBC 544 Input/Output and Interrupt Block Diagram

APPENDIX A (Continued)

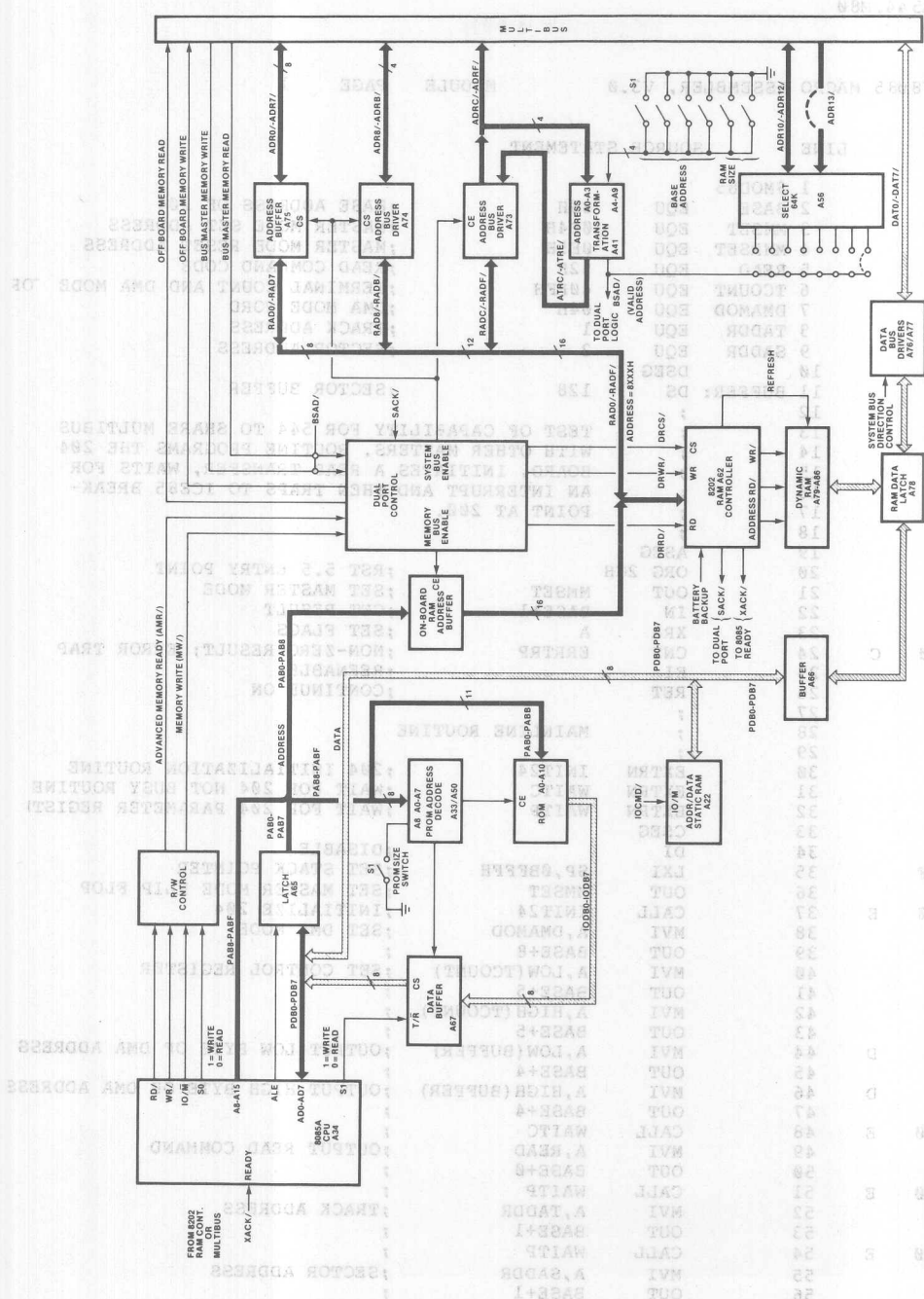


Figure A-2. ISBC 544 Memory Block Diagram

APPENDIX B

A M80 : F1:DMA544.M80

ISIS-II 8080/8085 MACRO ASSEMBLER, V3.0

MODULE PAGE 1

LOC	OBJ	LINE	SOURCE STATEMENT
		1	\$MOD85
0080		2	BASE EQU 80H ;BASE ADDRESS OF 204
00E4		3	MMSET EQU 0E4H ;MASTER MODE SET ADDRESS
00E5		4	MMRSET EQU 0E5H ;MASTER MODE RESET ADDRESS
0052		5	READ EQU 52H ;READ COMMAND CODE
40FF		6	TCOUNT EQU 40FFH ;TERMINAL COUNT AND DMA MODE 'OF
0004		7	DMAMOD EQU 04H ;DMA MODE WORD
0001		8	TADDR EQU 1 ;TRACK ADDRESS
0002		9	SADDR EQU 2 ;SECTOR ADDRESS
		10	DSEG
0000		11	BUFFER: DS 128 ;SECTOR BUFFER
		12	;
		13	TEST OF CAPABILITY FOR 544 TO SHARE MULTIBUS
		14	WITH OTHER MASTERS. ROUTINE PROGRAMS THE 204
		15	BOARD, INITIATES A READ TRANSFER, WAITS FOR
		16	AN INTERRUPT AND THEN TRAPS TO ICE85 BREAK-
		17	POINT AT 200.
		18	;
		19	ASEG
002C		20	ORG 2CH ;RST 5.5 ENTRY POINT
002C D3E4		21	OUT MMSET ;SET MASTER MODE
002E D881		22	IN BASE+1 ;GET RESULT
0030 AF		23	XRA A ;SET FLAGS
0031 C43900	C	24	CNZ ERRTRP ;NON-ZERO RESULT; ERROR TRAP
0034 FB		25	EI ;REENABLE
0035 C9		26	RET ;CONTINUE ON
		27	;
		28	;
		29	MAINLINE ROUTINE
		30	EXTRN INIT24 ;204 INITIALIZATION ROUTINE
		31	EXTRN WAITC ;WAIT FOR 204 NOT BUSY ROUTINE
		32	EXTRN WAITP ;WAIT FOR 204 PARAMETER REGIST
		33	CSEG
0000 F3		34	DI ;DISABLE
0001 31FFBF		35	LXI SP,0BFFFH ;SET STACK POINTER
0004 D3E4		36	OUT MMSET ;SET MASTER MODE FLIP FLOP
0006 CD0000	E	37	CALL INIT24 ;INITIALIZE 204
0009 3E04		38	MVI A,DMAMOD ;SET DMA MODE
000B D388		39	OUT BASE+8 ;
000D 3EFF		40	MVI A,LOW(TCOUNT) ;SET CONTROL REGISTER
000F D385		41	OUT BASE+5 ;
0011 3E40		42	MVI A,HIGH(TCOUNT) ;
0013 D385		43	OUT BASE+5 ;
0015 3E00	D	44	MVI A,LOW(BUFFER) ;OUTPUT LOW BYTE OF DMA ADDRESS
0017 D384		45	OUT BASE+4 ;
0019 3E00	D	46	MVI A,HIGH(BUFFER) ;OUTPUT HIGH BYTE OF DMA ADDRESS
001B D384		47	OUT BASE+4 ;
001D CD0000	E	48	CALL WAITC ;
0020 3E52		49	MVI A,READ ;OUTPUT READ COMMAND
0022 D380		50	OUT BASE+0 ;
0024 CD0000	E	51	CALL WAITP ;
0027 3E01		52	MVI A,TADDR ;TRACK ADDRESS
0029 D381		53	OUT BASE+1 ;
002B CD0000	E	54	CALL WAITP ;
002E 3E02		55	MVI A,SADDR ;SECTOR ADDRESS
0030 D381		56	OUT BASE+1 ;
0032 D3E5		57	OUT MMRSET ;RESET MASTER MODE FLIP/FLOP

APPENDIX B (Continued)

```

0034 FB          58          EI          ;ENABLE
0035 76          59          HLT         ;AND HALT          ; WAIT FOR INTEF
0036 C30002      60          JMP         200H ;TRAP TO ICE85 BREAKPOINT AT 200
          61  ERRTRP:      ;ERROR TRAP
0039 76          62          HLT         ;FOR NOW
          63          END

```

PUBLIC SYMBOLS

EXTERNAL SYMBOLS

```

INIT24 E 0000 WAITC E 0000 WAITP E 0000 U03 32AE 1 0000
      B22500A T32R 320E T32RAN: H420 U03 T32RM 1 1000
USER SYMBOLS: T32R 320E T32RAN: H420 U03 T32RM 1 1000
BASE A 0080 BUFFER D 0000 DMAMOD A 0004 ERRTRP C 0039 INIT24 E 0000 MF
READ A 0052 GMA SADDR A 0002 TADDR A 0001 TCOUNT A 40FF WAITC E 0000 WA

```

USER SYMBOLS

```
BASE    A 0080    BUFFER D 0000    DMAMOD A 0004    ERTRTP C 0039    INIT24 E 0000    MF
READ    A 0052    SADDR  A 0002    TADDR  A 0001    TCOUNT A 40FF    WAITC  E 0000    WAI
```

[illegible]

MODULE	PAGE	
	1	

APPENDIX B (Continued)

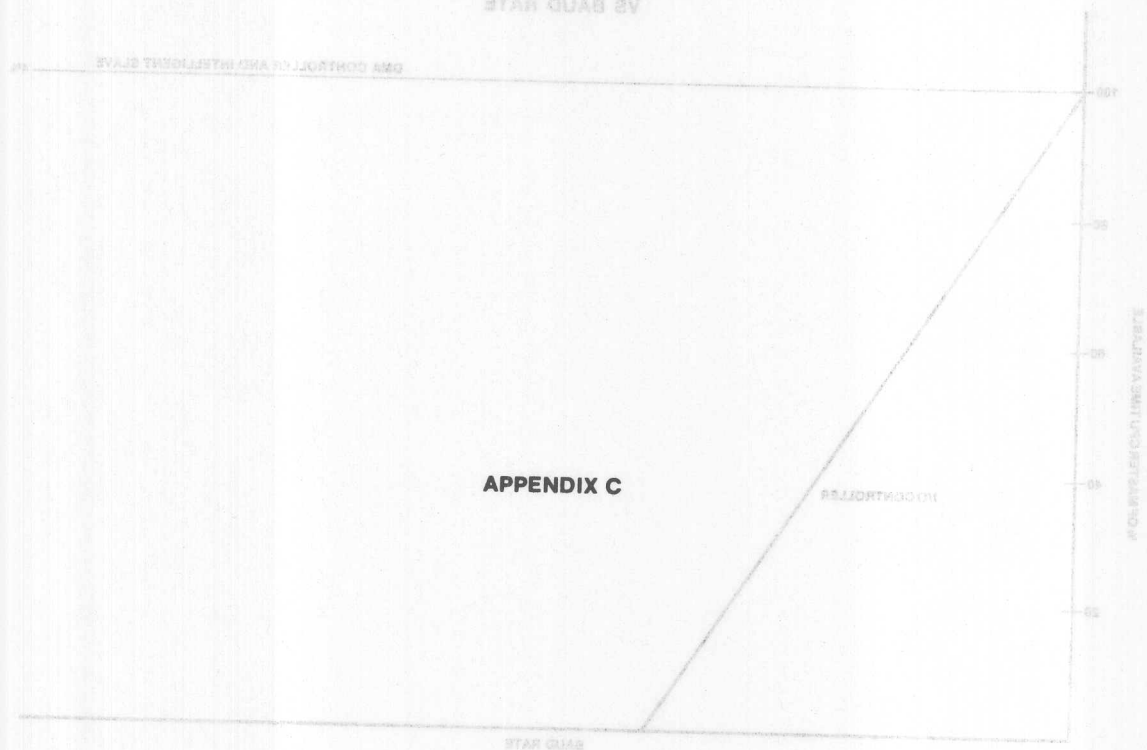
0035	3E35		55	MVI	A,SPECIFY	;SPECIFY BAD TRACKS
0037	D380		56	OUT	BASE+0	;
0039	CDA100	C	57	CALL	WAITP	;
003C	3E10		58	MVI	A,BADTR1	;BAD TRACKS FOR SURFACE 1
003E	D381		59	OUT	BASE+1	;
0040	CDA100	C	60	CALL	WAITP	;
0043	3EFF		61	MVI	A,NOBAD	;FIRST TRACK
0045	D381		62	OUT	BASE+1	;
0047	CDA100	C	63	CALL	WAITP	;
004A	3EFF		64	MVI	A,NOBAD	;SECOND BAD TRACK
004C	D381		65	OUT	BASE+1	;
004E	CDA100	C	66	CALL	WAITP	;
0051	3EFF		67	MVI	A,CTADDR	;CURRENT TRACK ADDRESS (NOT F'OW
0053	D381		68	OUT	BASE+1	;
0055	CD9900	C	69	CALL	WAITC	;
0058	3E35		70	MVI	A,SPECIFY	;
005A	D380		71	OUT	BASE+0	;
005C	CDA100	C	72	CALL	WAITP	;
005F	3E18		73	MVI	A,BADTR2	;SURFACE 2
0061	D381		74	OUT	BASE+1	;
0063	CDA100	C	75	CALL	WAITP	;
0066	3EFF		76	MVI	A,NOBAD	;FIRST TRACK
0068	D381		77	OUT	BASE+1	;
006A	CDA100	C	78	CALL	WAITP	;
006D	3EFF		79	MVI	A,NOBAD	;SECOND TRACK
006F	D381		80	OUT	BASE+1	;
0071	CDA100	C	81	CALL	WAITP	;
0074	3EFF		82	MVI	A,CTADDR	;CURRENT TRACK ADDRESS (NOT F'OW
0076	D381		83	OUT	BASE+1	;
0078	CD9900	C	84	CALL	WAITC	;
007B	3E69		85	MVI	A,SEEK	;SEEK TO TRACK 0
007D	D380		86	OUT	BASE+0	;
007F	CDA100	C	87	CALL	WAITP	;
0082	3E00		88	MVI	A,0	;
0084	D381		89	OUT	BASE+1	;
0086	D3E5		90	OUT	MMRSET	;GO TO SLEEP WHILE 204 DOES IT
0088	FB		91	EI		;ENABLE INTERRUPTS
0089	76		92	HLT		;SLEEP
008A	F3		93	DI		;DISABLE
008B	3E04		94	MVI	A,DMAMOD	;SET DMA MODE
008D	D388		95	OUT	BASE+8	;
008F	3E00		96	MVI	A,LOW(TCOUNT)	;SET CONTROL REGISTER
0091	D385		97	OUT	BASE+5	;
0093	3E40		98	MVI	A,HIGH(TCOUNT)	;
0095	D385		99	OUT	BASE+5	;
0097	FB		100	EI		;
0098	C9		101	RET		;RETURN
			102	;		
			103	;	WAITC AND WAITP ROUTINES	
			104	;		
0099	DB80		105	WAITC: IN	BASE+0	;GET STATUS BYTE
009B	E680		106	ANI	BUSY	;BUSY?
009D	C29900	C	107	JNZ	WAITC	;YES,LOOP
00A0	C9		108	RET		;NO,RETURN
			109	;		
00A1	DB80		110	WAITP: IN	BASE+0	;GET STATUS REGISTER
00A3	E620		111	ANI	PARFUL	;PARAMETER BUFFER FULL?
00A5	C2A100	C	112	JNZ	WAITP	;YES,LOOP
00A8	C9		113	RET		;NO,RETURN
			114	END		

PUBLIC SYMBOLS
INIT24 C 0000

WAITC C 0099 WAITP C 00A1

APPENDIX B (Continued)[illegible]

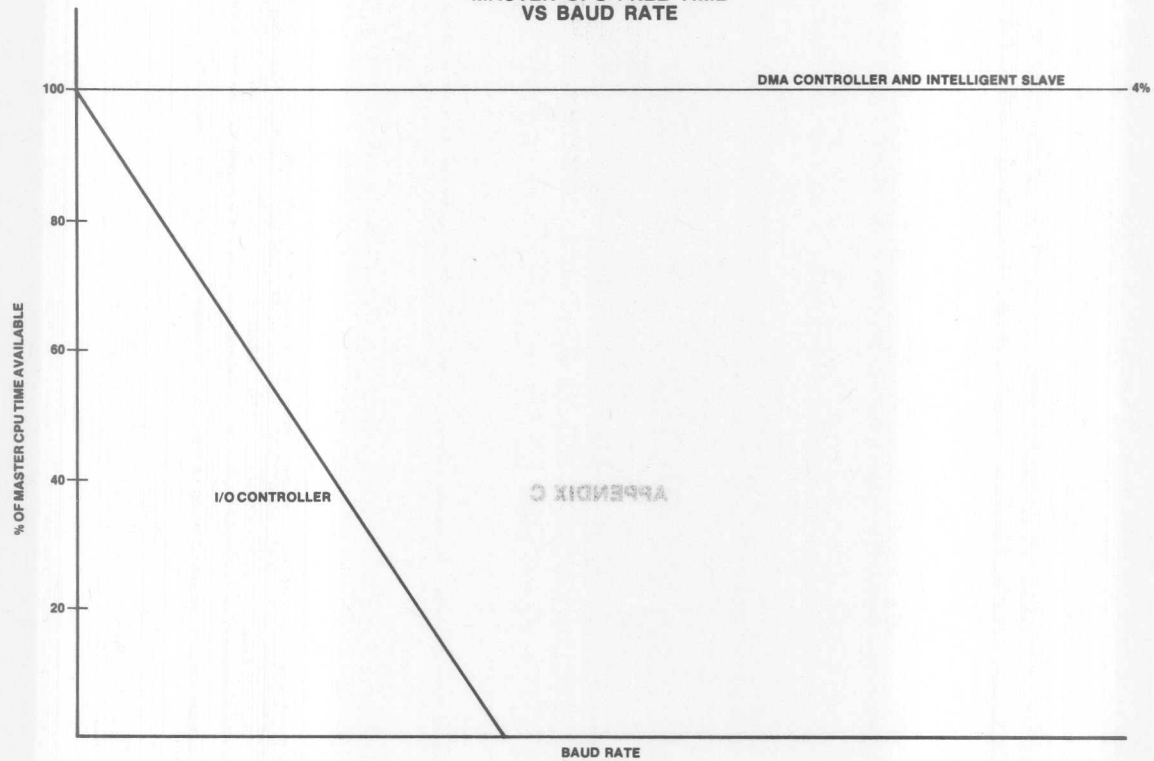
GRAPH 1
MASTER CPU FREE TIME
VS BAUD RATE



APPENDIX C

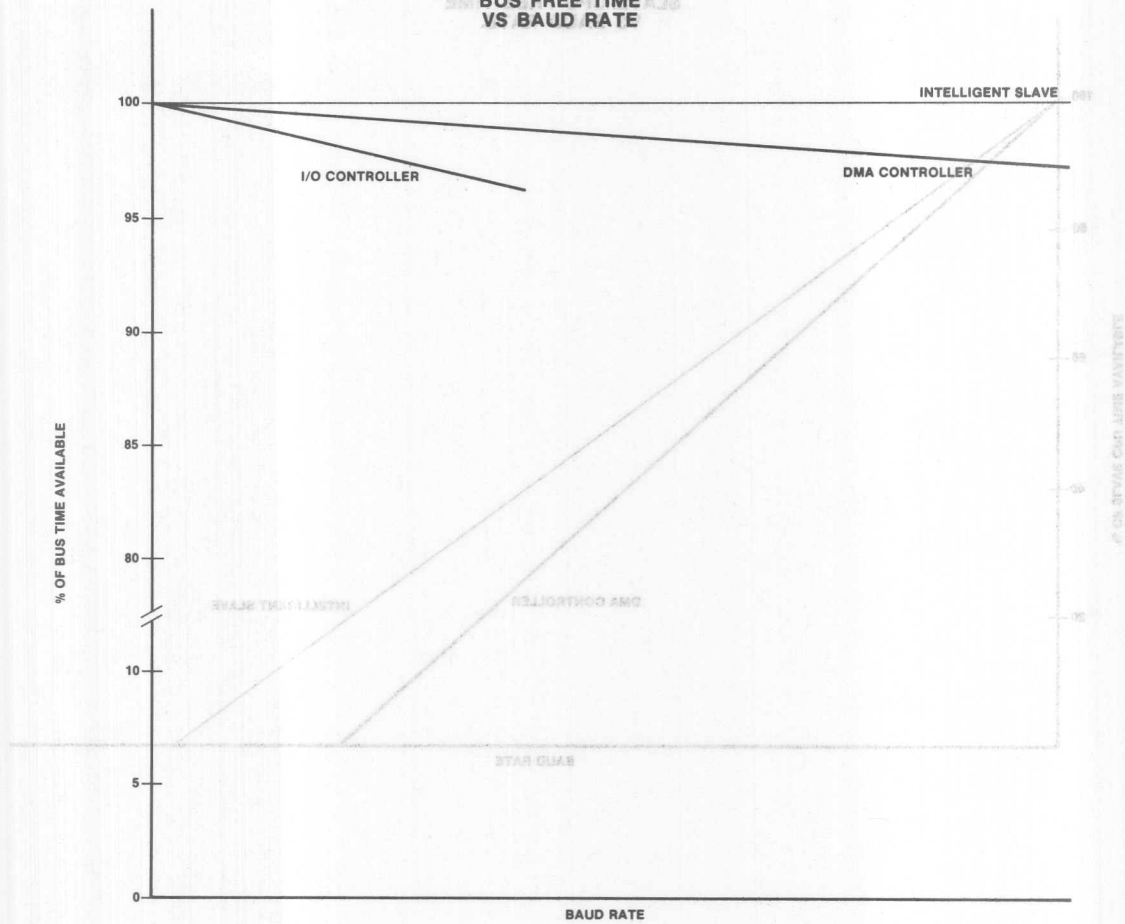
APPENDIX C (Continued)

GRAPH 1
MASTER CPU FREE TIME
VS BAUD RATE



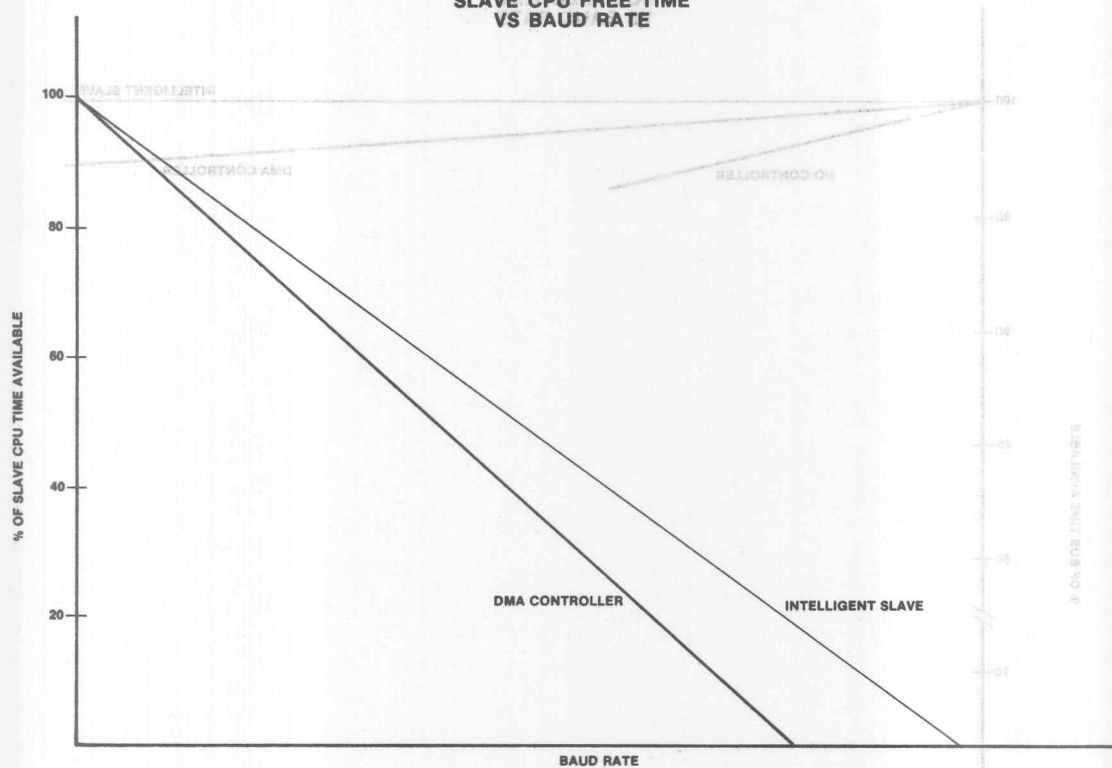
APPENDIX C (Continued)

GRAPH 2
BUS FREE TIME
VS BAUD RATE



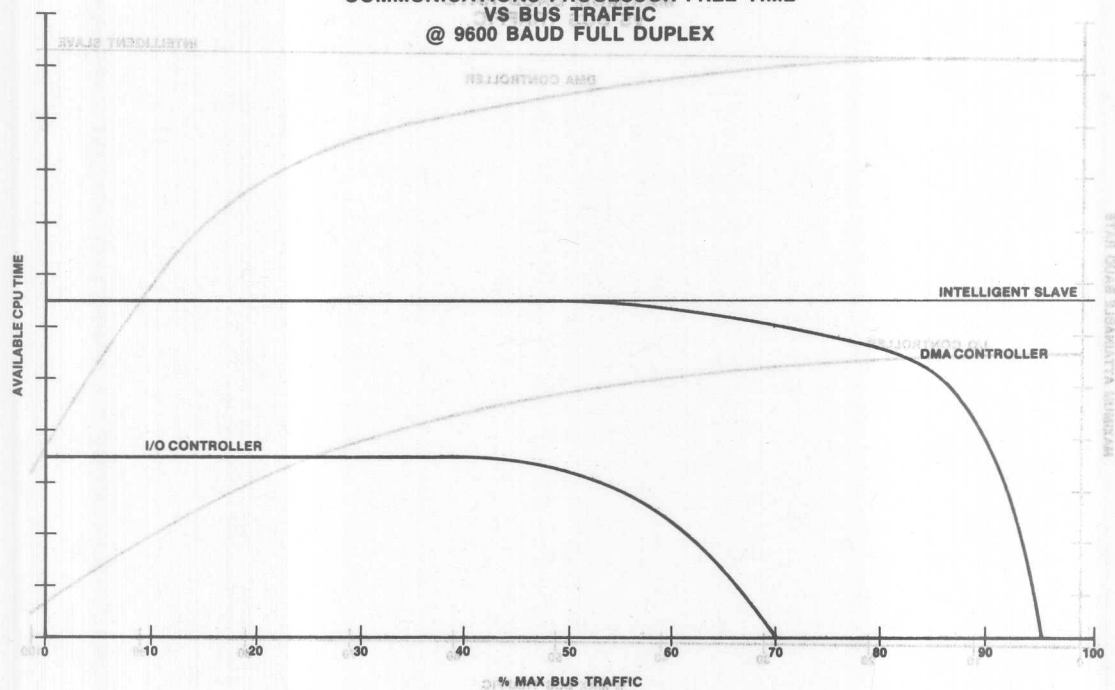
APPENDIX C (Continued)

GRAPH 3
SLAVE CPU FREE TIME
VS BAUD RATE

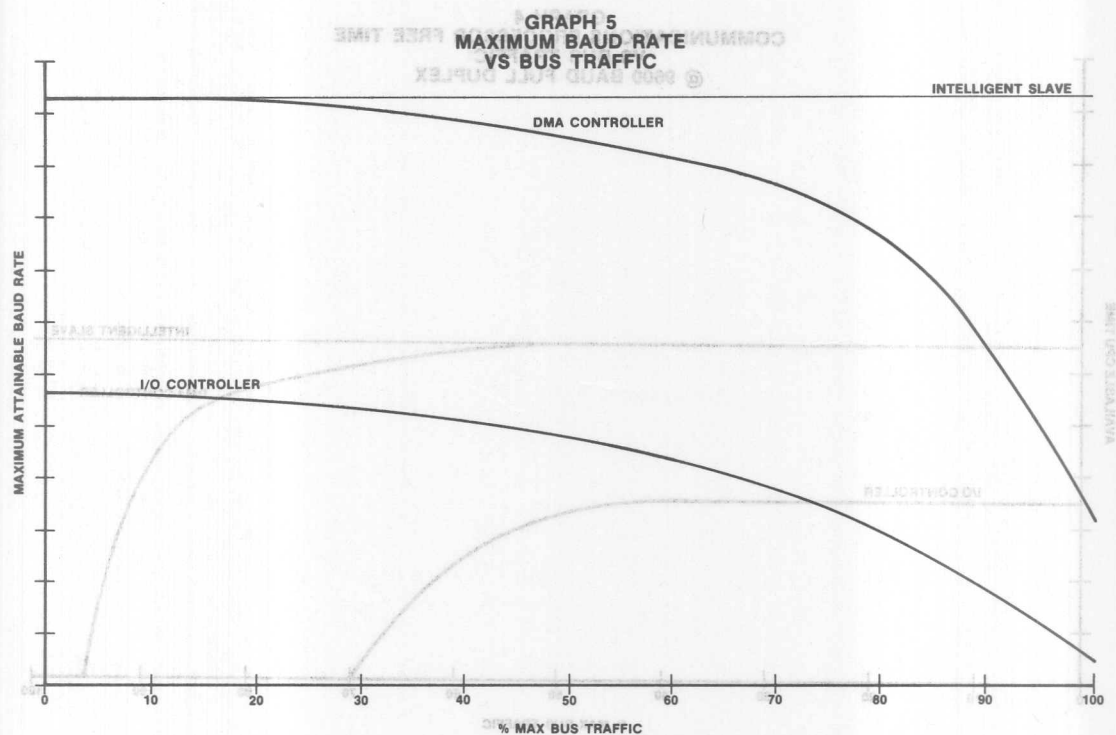


APPENDIX C (Continued)

GRAPH 4
COMMUNICATIONS PROCESSOR FREE TIME
VS BUS TRAFFIC
@ 9600 BAUD FULL DUPLEX



APPENDIX C (Continued)



(bound) APPENDIX D

PL/M-80 COMPILER SLAVE MAINLINE ROUTINE

PAGE 1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE MAINLINE
 OBJECT MODULE PLACED IN :F1:MAINLN.OBJ
 COMPILER INVOKED BY: PLM80 :F1:MAINLN.PLM PRINT(:F5:MAINLN.LST) PAGEWIDTH(78)

```

1      $title('slave mainline routine')
      main$line:
          DO;

/*
Mainline routine. Sets up stack$ptr, calls s$init to init-
ialize queues, initializes some of the hardware, sets up the
initial flag interrupt handlers, and then halts with interruptu
- pts
enabled allowing the rest of the system to operate totally
in interrupt mode.
*/

$noList

13      1.  initial$handler:  PROCEDURE EXTERNAL;
14      2.  END initial$handler;

15      1.  DECLARE
          command$word  LITERALLY  '4lh',
          port$a$8155  LITERALLY  '0e9h',
          command$8155  LITERALLY  '0e8h',
          mask$8259  LITERALLY  '0e7h',
          icw1$8259  LITERALLY  '0e6h',
          icw2$8259  LITERALLY  '0e7h',
          ocw3$8259  LITERALLY  '0e6h',
          read$ISR  LITERALLY  '0bh',
          mask$word  BYTE PUBLIC,
          port$a$value  BYTE PUBLIC,
          stat  BYTE,
          i  BYTE;

16      1.  output(icw1$8259)=0f6h;
17      1.  output(icw2$8259)=0fh;
18      1.  output(mask$8259),mask$word=0ffh;

19      1.  CALL s$init;

/* set up 8259 for ISR reads */

20      1.  output(ocw3$8259)=read$ISR;

21      1.  output(command$8155)=command$word;
22      1.  output(port$a$8155),port$a$value=0c0h;
23      1.  DO i=0 TO 7;
24      2.  stat=set$handler(i,.initial$handler);
  
```

APPENDIX D (Continued)

```

25 2      END;
26 1      DO WHILE 1;
27 2          HALT;
28 2      END;
29 1      END main$line;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 004DH      77D
VARIABLE AREA SIZE  = 0004H      4D
MAXIMUM STACK SIZE  = 0002H      2D
72 LINES READ
0 PROGRAM ERROR(S)

```

```

DECLARE
command$word LITERALLY '4th'
port$8155 LITERALLY '85h'
command$8155 LITERALLY '85h'
mask$8155 LITERALLY '85h'
icw1$8155 LITERALLY '85h'
icw2$8155 LITERALLY '85h'
ocw1$8155 LITERALLY '85h'
read$8155 LITERALLY '85h'
mask$word BYTE PUBLIC
port$8155 BYTE PUBLIC
start BYTE
i BYTE;

output(icw1$8155)=85h;
output(icw2$8155)=85h;
output(mask$8155,mask$word=85h);

CALL $init;

/* set up 8155 for ISR reads */
output(ocw1$8155)=read$8155;
output(command$8155)=command$word;
output(port$8155,port$8155=85h);
DO i=0 TO 7;
start=port$8155;

```

APPENDIX D (Continued)

P /M-80 COMPILER SLAVE APPLICATION LEVEL SIGNAL HANDLE

PAGE 1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INITIALHANDLER

OBJECT MODULE PLACED IN :F1:FINTRT.OBJ

COMPILER INVOKED BY: PLM80 :F1:FINTRT.PLM PRINT(:F5:FINTRT.LST) PAGESWIDTH(78)

```

1      $title('slave application level signal handler')
      initial$handler:
      DO;

/*
      Fields application level flag interrupts from the
      master. If the type=go$type the device attached to the queue
      specified is initialized with programming info sent into
      the queue by the master. If the type is data$available the
      specified transmitter is enabled unless a control$s pause
      is in effect.

*/
      $nolist

32 1  DECLARE
      no$pause LITERALLY '1',
      go$type LITERALLY '1',
      data$available LITERALLY '2',
      enable$xmit LITERALLY '1',
      reset LITERALLY '40h',
      timer$1$command$port LITERALLY '0dbh',
      timer$2$command$port LITERALLY '0dfh',
      mask$8259 LITERALLY '0e7h',
      mask$word BYTE EXTERNAL,
      mask (8) BYTE DATA(
          0fch,
          0fch,
          0f3h,
          0f3h,
          0cfh,
          0cfh,
          03fh,
          03fh),
      transmitter$state (8) BYTE PUBLIC,
      type BYTE,
      token BYTE,
      i BYTE,
      prog$info (5) BYTE,
      actual ADDRESS,
      usart$command$port (8) BYTE EXTERNAL,
      usart$state (8) BYTE PUBLIC,
      length$pointer (8) ADDRESS PUBLIC,
      pointer (8) ADDRESS PUBLIC,
      char$count (8) BYTE PUBLIC,
      timer$load$port (8) BYTE DATA(
          0d8h,

```


APPENDIX D (Continued)

```

PAGE 1
00d8h,
00d9h,
00da,
00dah,
00dah,
00dch,
00dch);

33 1 initial$handler: PROCEDURE (code) PUBLIC;
34 2 DECLARE code BYTE;
35 2 token=code AND 0fh;
36 2 type=shr (code,4);
37 2 IF type=go$type THEN
38 2 DO;
39 3 transmitter$state(token)=no$pause;
40 3 DO i=0 TO 3;
41 4 CALL varout(usart$command$port(token),0);
42 4 END;
43 3 CALL varout(usart$command$port(token),reset);
44 3 actual=get$line(token,.prog$info,5);
/* program the devices */
45 3 CALL varout(usart$command$port(token),prog$info(0));
46 3 CALL varout(usart$command$port(token),usart$state(token)
:=prog$info(1));
47 3 IF token < 7 THEN
48 3 CALL varout(timer$1$command$port,prog$info(2));
ELSE
49 3 CALL varout(timer$2$command$port,prog$info(2));
50 3 CALL varout(timer$load$port(token),prog$info(3));
51 3 CALL varout(timer$load$port(token),prog$info(4));
/* open up the four input queues for data input */
52 3 length$pointer(token-1)=new$line(token-1);
53 3 pointer(token-1)=next$space(token-1);
54 3 char$count(token-1)=0;
55 3 output(mask$8259),mask$word=mask$word AND mask(token);
56 3 END;
ELSE
57 2 IF (type=data$available) AND (transmitter$state(token)=no$pa
use) THEN
58 2 DO;
59 3 usart$state(token)=usart$state(token) OR enable$xmit;
60 3 CALL varout(usart$command$port(token),usart$state(token)
);
61 3 END;

```

APPENDIX D (Continued)

```

63      2      RETURN;
        END;
64      1      END initial$handler;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0182H      386D
VARIABLE AREA SIZE  = 0043H      67D
MAXIMUM STACK SIZE  = 0004H      4D
154 LINES READ
0 PROGRAM ERROR(S)

```

that a new line is available for processing. At this time the master is signalled to signify called to update the global queue pointer to permit access vacant when the line was started, and the exit primitive is encountered the length byte is filled in (it was left action is taken if any are found. When an engine character queue. A check is made for break characters and appropriate The character is read in and placed in the appropriate read to determine which device is requesting service. ready interrupt the 8255 in Service Request (SRQ) line. 544 resident interrupt service routine.

DECLARE

```

CONTROL$      LITERALLY '13H'
CONTROL$      LITERALLY '13H'
CONTROL$      LITERALLY '11H'
RUBOUT$       LITERALLY '7FH'
ESCAPE$        LITERALLY '13H'
CR            LITERALLY '0DH'
LF            LITERALLY '0AH'
BS            LITERALLY '08H'
SP            LITERALLY '20H'
BELL          LITERALLY '07H'
PTR           LITERALLY 'pointer(token)'
LENGTH$PTR    LITERALLY 'length$pointer(token)'
COUNT        LITERALLY 'char$count(token)'
DISABLE$EXIT   LITERALLY '07EH'
ENABLE$EXIT    LITERALLY '01H'
NO$PAUSE       LITERALLY '1'
PAUSE          LITERALLY '0'
LINE$AVAILABLE LITERALLY '1'
OCW$00000000  LITERALLY '000H'
OCW$00000001  LITERALLY '000H'
EOI            LITERALLY '10H'

```

DECLARE

```

VALUES$        ADDRESS
VALUE BASED VALUES$PTR BYTE
LINE$INDEXED$ BASED VALUES$PTR BYTE
DUMMY ADDRESS
ISR BYTE
TOKEN BYTE

```

APPENDIX D (Continued)

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INPUTHANDLER
 OBJECT MODULE PLACED IN :F1:INHDLR.OBJ
 COMPILER INVOKED BY: PLM80 :F1:INHDLR.PLM PRINT(:F5:INHDLR.LST) PAGEWIDTH(78)

```

1      $nointvector title('slave terminal input handler')
      input$handler:
          DO;

/*
544 resident interrupt service routine. After receiver
ready interrupt the 8259 In Service Register(ISR) is
read to determine which device is requesting service.
The character is read in and placed in the appropriate
queue. A check is made for break characters and appropriate
action is taken if any are found. When an endline character
is encountered the length byte is filled in ( it was left
vacant when the line was started) and the xmit primitive is
called to update the global queue pointer to permit access
to the line. At this time the master is signalled to signify
that a new line is available for processing.

*/

      $nolist

34      1      DECLARE
          control$x      LITERALLY      '18H',
          control$s      LITERALLY      '13H',
          control$q      LITERALLY      '11H',
          rubout          LITERALLY      '7FH',
          escape          LITERALLY      '1BH',
          CR              LITERALLY      '0DH',
          LF              LITERALLY      '0AH',
          BS              LITERALLY      '08H',
          SP              LITERALLY      '20H',
          bell            LITERALLY      '07H',
          ptr             LITERALLY      'pointer(token)',
          length$ptr      LITERALLY      'length$pointer(token)',
          count           LITERALLY      'char$count(token)',
          disable$xmit    LITERALLY      '0FEH',
          enable$xmit     LITERALLY      '01H',
          no$pause        LITERALLY      '1',
          pause           LITERALLY      '0',
          line$available  LITERALLY      '1',
          ocw2$8259       LITERALLY      '0E6H',
          ocw3$8259       LITERALLY      '0E6H',
          EOI             LITERALLY      '20H';

35      1      DECLARE
          value$ptr      ADDRESS,
          value BASED value$ptr BYTE,
          line$length BASED value$ptr BYTE,
          dummy          ADDRESS,
          ISR BYTE,
          token          BYTE,
  
```

APPENDIX D (Continued)

```

stat BYTE;
new$ptr ADDRESS;

36 1 DECLARE
    pointer (8) ADDRESS EXTERNAL,
    length$pointer (8) ADDRESS EXTERNAL,
    char$count (8) BYTE EXTERNAL,
    usart$state (8) BYTE EXTERNAL,
    usart$command$port (8) BYTE EXTERNAL,
    usart$data$port (8) BYTE EXTERNAL,
    transmitter$state (8) BYTE EXTERNAL;

37 1 index: PROCEDURE (value) BYTE;
38 2 DECLARE value BYTE;

39 2 IF value=control$x THEN RETURN 0;
41 2 IF value=rubout THEN RETURN 1;
43 2 IF value=control$s THEN RETURN 2;
45 2 IF value=control$q THEN RETURN 3;
47 2 IF value=escape THEN RETURN 4;
49 2 IF value=CR THEN RETURN 5;
51 2 RETURN 6;
52 2 END;

53 1 echo: PROCEDURE (token,buf$ptr,num$char) ADDRESS;
54 2 DECLARE (buf$ptr,num$char,actual) ADDRESS,
    token BYTE;

55 2 actual=send$line(token,buf$ptr,num$char);
56 2 usart$state(token)=usart$state(token) OR enable$xmit;
57 2 CALL varout(usart$command$port(token),usart$state(token));
58 2 RETURN actual;
59 2 END;

60 1 delete$line: PROCEDURE;

61 2 length$ptr=new$line(token);
62 2 ptr=next$space(token);
63 2 count=0;
64 2 dummy=echo(token+1,(' ',CR,LF),3);
65 2 RETURN;
66 2 END;

67 1 end$line: PROCEDURE;

68 2 value$ptr=length$ptr;
69 2 line$length=count;
70 2 ptr=next$space(token);
71 2 stat=xmit(token);
72 2 length$ptr=new$line(token);
73 2 ptr=next$space(token);
74 2 count=0;
75 2 stat=set$s$interrupt(token,line$available);
76 2 RETURN;
77 2 END;

```

```

78 1      in$hdr:      PROCEDURE INTERRUPT 0 PUBLIC;
79 2      ISR=input(ocw3$8259);

80 2      token=6;

81 2      again:
      ISR=shl(ISR,2);

82 2      IF NOT carry THEN
83 2          DO;
84 3              IF token=0 THEN RETURN; /* no bits set */
              ELSE
86 3                  DO;
87 4                      token=token-2;
88 4                      GOTO again;
89 4                  END;

90 3          END;
91 2      value$ptr=ptr;
92 2      value=varinp(usart$data$port(token)) AND 07fh;

93 2      DO CASE index(value);

          /* case 0; control$x; delete line */
94 3          DO;
95 4              CALL delete$line;
96 4          END;

          /* case 1; rubout; delete char */
97 3          DO;
98 4              new$ptr=back$space(token);
99 4              IF new$ptr=length$ptr THEN
100 4                  dummy=echo(token+1,.(bell),1);
              ELSE
101 4                  DO;
102 5                      dummy=echo(token+1,.(BS,SP,BS),3);
103 5                      ptr=new$ptr;
104 5                      count=count-1;
105 5                  END;
106 4              END;

          /* case 2; control$s; pause output */
107 3          DO;
108 4              usart$state(token+1)=usart$state(token+1) AND disabl
-      e$xmit;
109 4              CALL varout(usart$command$port(token+1),usart$state(
-      token+1));
110 4              transmitter$state(token+1)=pause;
111 4          END;

          /* case 3; control$q; resume output */
112 3          DO;
113 4              usart$state(token+1)=usart$state(token+1) OR enable$

```


(APPENDIX D (Continued))

```

PAGE 1
- xmit;
114 4 CALL varout(usart$command$port(token+1),usart$state(
- token+1));
115 4 transmitter$state(token+1)=no$pause;
116 4 END;
/* case 4; escape; terminate line */
117 3 DO;
118 4 dummy=echo(token+1,.( '#' ,CR,LF),3);
119 4 value=CR;
120 4 count=count+1;
121 4 CALL end$line;
122 4 END;
/* case 5; carriage return; terminate line */
123 3 DO;
124 4 dummy=echo(token+1,.(CR,LF),2);
125 4 count=count+1;
126 4 ptr=next$space(token);
127 4 value$ptr=ptr;
128 4 value=LF;
129 4 count=count+1;
130 4 CALL end$line;
131 4 END;
/* case 6; non-break character; stuff into queue */
132 3 DO;
133 4 dummy=echo(token+1,ptr,1);
134 4 ptr=next$space(token);
135 4 IF ptr=0 THEN CALL delete$line; /* full buffer */
137 4 ELSE count=count+1;
138 4 END;
139 3 END; /* of do case */
140 2 output(ocw3$8259)=EOI;
141 2 RETURN;
142 2 END;
143 1 END input$handler;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0398H    920D
VARIABLE AREA SIZE  = 0011H    17D
MAXIMUM STACK SIZE  = 0010H    16D
255 LINES READ
0 PROGRAM ERROR(S)

```

APPENDIX D (Continued)

P /M-80 COMPILER SLAVE CHARACTER OUTPUT HANDLER PAGE 1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE OUTPUTHANDLER
OBJECT MODULE PLACED IN :F1:OUTH.LR.OBJ
COMPILER INVOKED BY: PLM80 :F1:OUTH.LR.PLM PRINT(:F5:OUTH.LR.LST) PAGEWIDTH(78)

```

1      $nointvector title('slave character output handler')
      output$handler:
      DO;

/*
544 resident interrupt service routine. After transmitter
ready interrupt, 8259 In Service Register(ISR) is read to
determine which device is requesting service. A character
is requested from the appropriate queue and, if available,
is sent to the usart. If the queue is empty the transmitter
is disabled pending a signal from the master when more
characters are put into the queue.
*/

$noлист
11 1  DECLARE
      ocw2$8259 LITERALLY '0E6H',
      ocw3$8259 LITERALLY '0E6H',
      disable$xmit LITERALLY '0FEH',
      true LITERALLY '0FFH',
      false LITERALLY '00H',
      EOI LITERALLY '0A0H';

12 1  DECLARE
      ISR BYTE,
      token BYTE,
      actual ADDRESS,
      value BYTE;

13 1  DECLARE
      usart$state (8) BYTE EXTERNAL,
      usart$command$port (8) BYTE PUBLIC DATA(
          0D1H,
          0D1H,
          0D3H,
          0D3H,
          0D5H,
          0D5H,
          0D7H,
          0D7H),
      usart$data$port (8) BYTE PUBLIC DATA(
          0D0H,
          0D0H,
          0D2H,
          0D2H,
          0D4H,

```

APPENDIX D (Continued)

```

PAGE 1                                00D4H,
                                      00D6H,
                                      00D6H);
14 1      out$hlr:      PROCEDURE INTERRUPT 1 PUBLIC;
                                /* get active level number and use it to determine queue$token */
                                - /
15 2          ISR=input(ocw3$8259);
16 2          token=7;
17 2      again:
                                ISR=shl(ISR,1);
18 2          IF NOT carry THEN
19 2              DO;
20 3              IF token=1 THEN RETURN; /* no bits in ISR set */
                                ELSE
22 3                  DO;
23 4                      token=token-2;
24 4                      ISR=shl(ISR,1);
25 4                      GOTO again;
26 4                  END;
27 3          END;
28 2          actual=get$line(token,.value,1);
29 2          IF actual=0 THEN
30 2              DO; /* empty queue. Disable transmitter */
31 3              usart$state(token)=usart$state(token) AND disabl
                                - e$xmtr;
32 3              CALL varout(usart$command$port(token),usart$stat
                                - e(token));
33 3              END;
                                ELSE
34 2              CALL varout(usart$data$port(token),value);
35 2              output(ocw3$8259)=EOI;
36 2              RETURN;
37 2          END;
38 1      END output$handler;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 00A4H      164D
VARIABLE AREA SIZE  = 0005H       5D
MAXIMUM STACK SIZE  = 000CH      12D
102 LINES READ
0 PROGRAM ERROR(S)

```

APPENDIX D (Continued)

PL/M-80 COMPILER RMX/80-544 INITIALIZATION TASK PAGE 1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INIT544
 OBJECT MODULE PLACED IN :F1:INIT54.OBJ
 COMPILER INVOKED BY: PLM80 :F1:INIT54.PLM PRINT(:F5:INIT54.LST) PAGEWIDTH(78)

```

1      $title('rmx/80-544 initialization task')
      init$544:
      DO;

/*
Task code for 544 driver initialization task. Info
from application supplied memory allocation block
is accessed to set up queues and transfer device programming
info to the slave board(s) and create the required
service handler tasks.
*/

$noList

56 1    input$driver:  PROCEDURE EXTERNAL;
57 2    END input$driver;

58 1    output$driver: PROCEDURE EXTERNAL;
59 2    END output$driver;

60 1    signal: PROCEDURE EXTERNAL;
61 2    END signal;

62 1    DECLARE
        stack$size  LITERALLY '256',
        go$type     LITERALLY '1';

63 1    DECLARE
        ptr         ADDRESS,
        init$stable BASED ptr STRUCTURE(
            base$adr ADDRESS,
            queue$token BYTE,
            prog$info (5) BYTE),
        i           BYTE,
        overflow    ADDRESS,
        queue$init$stable (1) STRUCTURE(
            base$adr ADDRESS,
            queue$size (8) ADDRESS) EXTERNAL,
        initialization$stable (1) BYTE EXTERNAL,
        stat        BYTE,
        num$devices  BYTE EXTERNAL,
        num$boards  BYTE EXTERNAL,
        service$exchange$stable (1) ADDRESS EXTERNAL,
        signal$exchange$stable (1) ADDRESS EXTERNAL,
        service$exchanges (1) BYTE EXTERNAL,
        signal$exchanges (1) BYTE EXTERNAL,
        task$descriptors (1) BYTE EXTERNAL,

```

APPENDIX D (Continued)

```

stacks (1) BYTE EXTERNAL,
info$block (1) STRUCTURE
    /* create service and queue$token BYTE,
    index BYTE) EXTERNAL,
    rgactv ADDRESS EXTERNAL;
64 1 DECLARE
    rom$input$std static$task$descriptor DATA(
        'input',
        .input$driver,
        0, /* stack will be assigned individually */
        stack$size,
        200,
        0, /* tba */
        0), /* tba */
    rom$output$std static$task$descriptor DATA(
        'output',
        .output$driver,
        0,
        stack$size,
        201,
        0,
        0),
    input$hdlr$std static$task$descriptor,
    output$hdlr$std static$task$descriptor;
65 1 init$xch: PROCEDURE (xch$ptr);
    /* initializes expanded interrupt exchanges */
66 2 DECLARE xch$ptr ADDRESS,
    xch BASED xch$ptr int$exchange$descriptor;
67 2 xch.link=.xch.link;
68 2 xch.type=int$type;
69 2 xch.length=5;
70 2 RETURN;
71 2 END;
72 1 init$54: PROCEDURE PUBLIC;
73 2 DO i=0 TO num$boards-1;
74 3 CALL m$init(.queue$init$table(i));
75 3 END;
76 2 CALL move(size(rom$input$std),.rom$input$std,.input$hdlr$std
- );
77 2 CALL move(size(rom$output$std),.rom$output$std,.output$hdlr$
- std);
78 2 ptr=.initialization$table;
79 2 DO i=0 TO num$devices*2 BY 2;
/* send programming info to slave */
80 3 overflow=send$line(init$table.base$adr,init$table.queue$
- token,.init$table.prog$info,5);
81 3 stat=set$m$interrupt(init$table.base$adr,init$table.queu

```


APPENDIX D (Continued)

```

- e$token,go$type);
/* create service and signal exchanges */
82 3 CALL rqcxcx(service$exchange$table(i):=.service$exchange
- s+10*i);
83 3 CALL rqcxcx(service$exchange$table(i+1):=.service$exchan
- ges+10*(i+1));
84 3 CALL init$xcx(.signal$exchanges+15*i);
85 3 CALL init$xcx(.signal$exchanges+15*(i+1));
86 3 CALL rqcxcx(signal$exchange$table(i):=.signal$exchanges+
- 15*i);
87 3 CALL rqcxcx(signal$exchange$table(i+1):=.signal$exchange
- s+15*(i+1));
88 3 info$block(i).base$adr;
89 3 info$block(i+1).base$adr=init$table.base$adr;
90 3 info$block(i).queue$token=init$table.queue$token-1;
91 3 info$block(i+1).queue$token=init$table.queue$token;
92 3 info$block(i).index=i;
93 3 info$block(i+1).index=i+1;
94 3 input$hdlr$std.sp=.stacks+stack$size*i;
95 3 output$hdlr$std.sp=.stacks+stack$size*(i+1);
96 3 input$hdlr$std.exchange$address=.info$block(i);
97 3 output$hdlr$std.exchange$address=.info$block(i+1);
98 3 input$hdlr$std.task$ptr=.task$descriptors+20*i;
99 3 output$hdlr$std.task$ptr=.task$descriptors+20*(i+1);
100 3 CALL rqctsk(.input$hdlr$std);
101 3 CALL rqctsk(.output$hdlr$std);
102 3 ptr=ptr+8;
END;
103 2 CALL rqsetv(.signal,2);
104 2 CALL rqelvl(2);
105 2 CALL rqsusp(rqactv);
106 2 END; /* of task */
107 1 END init$544;

```

MODULE INFORMATION:

```

CODE AREA SIZE=02C3H 707Dn
VARIABLE AREA SIZE = 002AH 42D
MAXIMUM STACK SIZE=0006H 6D
285 LINES READ
0 PROGRAM ERROR(S)

```

APPENDIX D (Continued)

P /M-80 COMPILER RMX/80-544 INITIALIZATION MODULE AND

PAGE 1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INITMODULE

OBJECT MODULE PLACED IN :F1:MAB.OBJ

COMPILER INVOKED BY: PLM80 :F1:MAB.PLM PRINT(:F5:MAB.LST) PAGEWIDTH(78)

```

1      $title('rmx/80-544 initialization module and memory allocation b
      - lock')
1      init$module:
          DO;

      /*
      - 44      Initialization tables created and allocation of memory for 5
          handler done here.
      */

2      1      DECLARE
          number$of$devices LITERALLY '4',
          baud$rate$count$1 LITERALLY '32',
          baud$rate$count$h LITERALLY '00',
          usart$mode LITERALLY '4eh',
          usart$cmd LITERALLY '27h',
          ctr$0$mode LITERALLY '36h',
          ctr$1$mode LITERALLY '76h',
          ctr$2$mode LITERALLY '0b6h',
          ctr$3$mode LITERALLY '36h',
          num$devices BYTE PUBLIC DATA(number$of$devices-1),
          num$boards BYTE PUBLIC DATA(1),
          service$exchange$table(8) ADDRESS PUBLIC,
          signal$exchange$table(8) ADDRESS PUBLIC,
          signal$type(8) BYTE PUBLIC,
          service$exchanges(80) BYTE PUBLIC,
          signal$exchanges(120) BYTE PUBLIC,
          task$descriptors(160) BYTE PUBLIC,
          stacks(2048) BYTE PUBLIC,
          info$block(32) BYTE PUBLIC,
          queue$init$table(1) STRUCTURE(
              base$adr ADDRESS,
              queue$size(8) ADDRESS) PUBLIC DATA(
                  0e000h,
                  256,
                  1765,
                  256,
                  1765,
                  256,
                  1765,
                  256,
                  1765,
                  1765),
          base$table(1) ADDRESS PUBLIC DATA(
              0e000h),
          initialization$table(number$of$devices) STRUCTURE(
              base$adr ADDRESS,

```

APPENDIX D (Continued)

MODULE INFORMATION:

```
CODE AREA SIZE      = 0036H      54D
VARIABLE AREA SIZE  = 09B0H      2480D
MAXIMUM STACK SIZE  = 0000H       0D
79 LINES READ
0 PROGRAM ERROR(S)
```

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE SIGNALHANDLER
 OBJECT MODULE PLACED IN :F1:SIGNAL.OBJ
 COMPILER INVOKED BY: PLM80 :F1:SIGNAL.PLM PRINT(:F5:SIGNAL.LST) PAGEWIDTH(78)

```

1      $nointvector title('slave->master interrupt handler')
      signal$handler:
          DO;

/*
Fields all slave->master signals(interrupts) and calls rqisn
-   d
with the proper signal exchange address.
*/

$no!ist

26  1      DECLARE
          i          BYTE,
          ptr        ADDRESS,
          (flag BASED ptr) BYTE,
          num$boards BYTE EXTERNAL,
          num$devices BYTE EXTERNAL,
          signal$type (1) BYTE EXTERNAL,
          index      BYTE,
          token       BYTE,
          signal$exchange$table (1) ADDRESS EXTERNAL,
          base$table (1) ADDRESS EXTERNAL;

27  1      signal: PROCEDURE INTERRUPT 2 PUBLIC;

/* poll slave boards and find one generating interrupt */

28  2          i=0;

29  2      next:
          ptr=base$table(i)+1;
30  2          IF flag=0 THEN
31  2              DO;
32  3              i=i+1;
33  3              IF i > num$boards THEN RETURN; /* erroneous signal *
-   /
35  3              ELSE GOTO next;
36  3          END;

/* get queue token and use it to index into signal exchange tabl
-   e */

37  2          token=(flag AND 0fh);
38  2          index=4*i+token;

/* if index is out of range don't attempt the isend */

```

APPENDIX D (Continued)

```

39 2 IF index <= num$devices THEN
40 2 DO;
41 3 CALL rqisnd(signal$exchange$table(index));
42 3 signal$type(index)=shr(flag,4);
43 3 END;
ELSE
44 2 CALL rqendi;

/* zero flag to acknowledge interrupt */

45 2 flag=0;
46 2 RETURN;
47 2 END;
48 1 END signal$handler;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 008BH      139D
VARIABLE AREA SIZE  = 0005H      5D
MAXIMUM STACK SIZE  = 000AH      10D
110 LINES READ
0 PROGRAM ERROR(S)

```

```

DECLARE
  i
  ptr
  (flag BASED ptr) BYTE
  num$boards BYTE EXTERNAL
  num$devices BYTE EXTERNAL
  signal$type (i) BYTE EXTERNAL
  index
  token
  signal$exchange$table (i) ADDRESS EXTERNAL
  base$table (i) ADDRESS EXTERNAL

  signal: PROCEDURE INTERRUPT 2 PUBLIC;

/* poll slave boards and find one generating interrupt */
  i=0;
  next:
    ptr=base$table(i)+1;
    IF flag=8 THEN
      DO;
        i=i+1;
        IF i > num$boards THEN RETURN; /* erroneous signal */
      ELSE GOTO next;
    END;

/* get queue token and use it to index into signal exchange table
  token=(flag AND 8);
  index=i+token;

/* if index is out of range don't attempt the read */

```


APPENDIX D (Continued)

P /M-80 COMPILER RMX/80-544 INPUT SERVICE HANDLER PAGE 32 1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INPUTDRIVER

OBJECT MODULE PLACED IN :F1:INPUT.OBJ

COMPILER INVOKED BY: PLM80 :F1:INPUT.PLM PRINT(:F5:INPUT.LST) PAGERWIDTH(78)

```

1      $title('rmx/80-544 input service handler')
      input$driver:
      DO:
/*
/* Master resident task code. Monitors service exchange
and fills input requests by retrieving characters from
the proper queue(board$base and device info is passed
via default exchange field). By definition the first byte
of a line of input contains the length of that line.
This figure is used to retrieve the exact number of character
rs
available in a given line.
*/
$noinput
27 1      DECLARE
          rqactv ADDRESS EXTERNAL,
          td BASED rqactv task$descriptor,
          service$exchange$table (1) ADDRESS EXTERNAL,
          signal$exchange$table (1) ADDRESS EXTERNAL;
28 1      input$driver: PROCEDURE REENTRANT PUBLIC;
29 2      DECLARE
          service$exchange ADDRESS,
          board$base ADDRESS,
          queue$token BYTE,
          signal$exchange ADDRESS,
          msg$ptr ADDRESS,
          msg BASED msg$ptr th$msg,
          actual ADDRESS,
          dummy ADDRESS,
          info$block$ptr ADDRESS,
          info$block BASED info$block$ptr STRUCTURE(
              base$adr ADDRESS,
              queue$token BYTE,
              index BYTE),
          num$char BYTE,
          stat BYTE;

/* get info out of default field */
30 2      info$block$ptr=td.exchange$address; /* default exchange field
31 2      service$exchange=service$exchange$table(info$block.index);

```

APPENDIX D (Continued)

```

32 2 board$base=info$block.base$adr;
33 2 queue$token=info$block.queue$token;
34 2 signal$exchange=signal$exchange$stable(info$block.index);
35 2 DO forever;

/* wait for request message */

36 3 msg$ptr=rqwait(service$exchange,0);

37 3 retry:
/* try to get line count out of queue */
actual=get$line(board$base,queue$token,.num$char,1);

/* if unsuccessful wait for signal and try again */
38 3 IF actual=0 THEN
39 3 DO;
40 4 dummy=rqwait(signal$exchange,0);
41 4 GOTO retry;
42 4 END;

/* if all okay get line */
43 3 actual=get$line(board$base,queue$token,msg.buffer$adr,num
- m$char);
44 3 msg.actual=actual;
45 3 msg.status=0;
46 3 CALL rgsend(msg.resp$ex,msg$ptr);
47 3 END; /* of do forever */
48 2 END; /* of task */
49 1 END input$driver;

```

MODULE INFORMATION:

```

CODE AREA SIZE = 012CH 300D
VARIABLE AREA SIZE = 0000H 0D
MAXIMUM STACK SIZE = 0017H 23D
171 LINES READ
0 PROGRAM ERROR(S)

```

```

/* get info out of default field */
info$block$ptr=td.exchange$address; /* default exchange field
/*
service$exchange=service$exchange$stable(info$block.index);

```

APPENDIX D (Continued)

P /M-80 COMPILER RMX/80-544 OUTPUT SERVICE HANDLER PAGE 1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE OUTPUTDRIVER
OBJECT MODULE PLACED IN :F1:OUTPUT.OBJ
COMPILER INVOKED BY: PLM80 :F1:OUTPUT.PLM PRINT(:F5:OUTPUT.LST) PAGERWIDTH(78)

```

1      $title('rmx/80-544 output service handler')
      output$driver:
      DO;

/*
Master resident task code. Monitors service exchange and
fills output requests by stuffing characters into the appropriate
queue. If insufficient room is available the task waits
for 1 second and retries up to 100 times after which it
signals a time out error. If the transmission completes
successfully the slave is signalled to indicate that data is
available.
*/
$olist

39 1      DECLARE
          data$available LITERALLY '2',
          time$out LITERALLY '1';

40 1      DECLARE
          rqactv ADDRESS EXTERNAL,
          (td BASED rqactv) task$descriptor,
          service$exchange$table (1) ADDRESS EXTERNAL,
          signal$exchange$table (1) ADDRESS EXTERNAL;

41 1      output$driver: PROCEDURE REENTRANT PUBLIC;

42 2      DECLARE
          service$exchange ADDRESS,
          signal$exchange ADDRESS,
          base$adr ADDRESS,
          queue$token BYTE,
          msg$ptr ADDRESS,
          msg BASED msg$ptr th$msg,
          tries$left BYTE,
          overflow ADDRESS,
          dummy ADDRESS,
          stat BYTE,
          info$block$ptr ADDRESS,
          info$block BASED info$block$ptr STRUCTURE(
              base$adr ADDRESS,
              queue$token BYTE,
              index BYTE);

```

APPENDIX D (Continued)

```

1      PAGE      /* initialize */
43     2          info$block$ptr=td.exchange$address;
44     2          service$exchange=service$exchange$stable(info$block.index);
45     2          signal$exchange=signal$exchange$stable(info$block.index);
46     2          base$adr=info$block.base$adr;
47     2          queue$token=info$block.queue$token;

48     2          DO forever;
          /* wait for request message */

49     3          msg$ptr=rqwait(service$exchange,0);
50     3          tries$left=100;
51     3  retry:  overflow=send$line(base$adr,queue$token,msg.buffer$adr,m
          - sg.count);
52     3          IF overflow <> 0 THEN
53     3              DO;
54     4              dummy=rqwait(signal$exchange,20);
55     4              tries$left=tries$left-1;
56     4              IF tries$left > 0 THEN GOTO retry;
          ELSE
58     4              DO;
59     5                  msg.status=time$out;
60     5                  msg.actual=0;
61     5                  GOTO quit;
62     5              END;
63     4          END;
64     3          msg.status=0;
65     3          stat=set$m$interrupt(base$adr,queue$token,data$available
          - );
66     3          msg.actual=msg.count;
67     3  quit:  CALL rgsend(msg.resp$ex,msg$ptr);
68     3          END; /* of do forever */

69     2          END; /* of task */

70     1          END output$driver;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0159H 345D
VARIABLE AREA SIZE  = 0000H 0D
MAXIMUM STACK SIZE  = 0019H 25D
198 LINES READ
0 PROGRAM ERROR(S)

```

APPENDIX D (Continued)

A M80 : F1:CFG544.M80 PRINT(:F4:CFG544.LST) PAGEWIDTH(78) MACROFILE

ISIS-II 8080/8085 MACRO ASSEMBLER, V3.0

CFG544 PAGE 1

LOC	OBJ	LINE	SOURCE STATEMENT
		1	NAME CFG544
		2	CSEG
		3	PUBLIC RQRATE
0000 0800		4	RQRATE: DW 8
		5	\$NOLIST
		127	\$LIST
		128	\$NOGEN
0000		129	NTASK SET 0
0000		130	NEXCH SET 0
		131	;
		132	; BUILD THE INITIAL TASK TABLE
		133	;
		134	; -----\ THIS TASK IS NECESSARY FOR THE 544 HANDLE
		R	
		135	; -----/ IT CREATES EVERYTHING ELSE IT NEEDS.
		136	STD INIT54,200,1,0
		191	STD LINECH,64,130,0
		246	
		247	; ALLOCATE TASK DESCRIPTORS
		248	;
		249	GENTD
		253	;
		254	; BUILD INITIAL EXCHANGE TABLE
		255	;
		256	XCHADR RESPEX
		264	;
		265	; BUILD CREATE TABLE
		266	;
		267	CRTAB
		274	END

PUBLIC SYMBOLS

RQCRTB C 0026 RQRATE C 0000

EXTERNAL SYMBOLS

INIT54 E 0000 LINECH E 0000 RESPEX E 0000

USER SYMBOLS

CRTAB + 0000	GENTD + 0000	IET C 0024	INIT54 E 0000
INTXCH + 0000	ITT C 0002	LINECH E 0000	NEXCH A 0001
NTASK A 0002	PUBXCH + 0007	RESPEX E 0000	RQCRTB C 0026
RQRATE C 0000	STD + 0000	TDBASE D 0108	XCH + 0005
XCHADR + 0002			

1-177	INTRODUCTION
1-177	isBX™ MULTIMODULE™ BOARD CONCEPT
1-178	isBX™ MULTIMODULE™ SYSTEM INTERFACE
1-178	Host Boards
1-178	isBX™ MULTIMODULE™ Boards
1-180	isBX™ Connector
1-180	isBX™ Bus Interface Signals

July 1980

1-181	isBX™ BUS INTERFACING
1-181	Bus Timing
1-181	Command Operations
1-183	isBX™ Addressing
1-183	Considerations for isBX™ Bus Interfacing
1-184	Optional Interface Lines
1-185	isBX™ MULTIMODULE™ DESIGN EXAMPLE
1-185	isBX™ MULTIMODULE™ Board Design
1-186	Display Module Design
1-187	Keyboard Interface Design
1-187	Operation With The iSBC 80V10B™
1-187	Single Board Computer
1-187	Breadboarding The Design
1-189	Software Considerations
1-190	Debug Considerations
1-190	SUMMARY
1-192	APPENDIX A — isBX™ Signals Pin Assignments
1-193	APPENDIX B — isBX™ MULTIMODULE™ I/O AC Specifications
1-194	APPENDIX C — Listing for the isBX™ Design Example Software Exerciser

Designing isBX™ MULTIMODULE™ Boards

Stephen Grubb
OEM Microcomputer
Systems Applications

Designing iSBX™ MULTIMODULE™ Boards

Contents

INTRODUCTION	1-177
iSBX™ MULTIMODULE™ BOARD CONCEPT	1-177
iSBX™ MULTIMODULE™ SYSTEM INTERFACE	1-178
Host Boards	1-178
iSBX™ MULTIMODULE™ Boards	1-178
iSBX™ Connector	1-180
iSBX™ Bus Interface Signals	1-180
iSBX™ BUS INTERFACING	1-181
Bus Timing	1-181
Command Operations	1-181
iSBX™ Addressing	1-183
Considerations for iSBX™ Bus Interfacing	1-183
Optional Interface Lines	1-184
iSBX™ MULTIMODULE™ DESIGN EXAMPLE	1-185
iSBX™ MULTIMODULE™ Board Design	1-185
Display Module Design	1-186
Keyboard Interface Design	1-187
Operation With The iSBC 80/10B™ Single Board Computer	1-188
Breadboarding The Design	1-189
Software Considerations	1-189
Debug Considerations	1-190
SUMMARY	1-190
APPENDIX A — iSBX™ Signal Pin Assignments	1-192
APPENDIX B — iSBX™ MULTIMODULE™ I/O AC Specifications	1-193
APPENDIX C — Listing for the iSBX™ Design Example Software Exerciser	1-194

INTRODUCTION

Intel's single board computers and the MULTIBUS™ system bus have become de facto industry standards in the microcomputer board market. The speed and capability of the bus coupled with the functionality and performance of the boards have been used to solve a large number of problems. iSBC products are in applications ranging from simple single board relay replacement to sophisticated multi-board business systems supporting large hard disk files. However, even with the range of functionality provided by standard iSBCs and expansion boards, designers have felt the need to design custom MULTIBUS-compatible boards to fit their application. Until the introduction of the iSBX concept, these custom boards had to be implemented using a separate MULTIBUS form factor board.

Intel has recently introduced a new line of board products and a new bus which are destined to become another industry standard because of the niche they fill. The new iSBX MULTIMODULE boards are designed to extend the functional capabilities of single board computers at a much lower cost than previously possible. iSBX MULTIMODULE boards are supported by a new bus — the iSBX bus, which allows the MULTIMODULE boards to be added directly to the on-board microprocessor bus. iSBX MULTIMODULE boards are from 10 to 20 square inches in size, therefore permitting small modular increments to a single board computer's capabilities.

System designers now have the capabilities of using either standard iSBCs or iSBX MULTIMODULE boards, or designing custom MULTIBUS compatible or iSBX MULTIMODULE boards. Cost-effective solutions are easily realized because of this added flexibility.

This application note discusses the iSBX MULTIMODULE concept, currently available MULTIMODULE boards and the iSBCs which support these boards. The iSBX bus interface specifications are discussed next, followed by consideration for designing custom iSBX MULTIMODULE boards. A specific design example using an Intel® 8279 Programmable Keyboard/Display Controller is presented.

The objective of the note is to introduce the reader to the iSBX MULTIMODULE concept for expanding iSBC functionality and to illustrate how a designer can effectively use this concept with either standard or custom iSBX boards.

References to further documentation on the iSBX bus, specific iSBX MULTIMODULE boards and iSBC host boards currently available may be found in the Related Intel Publications section in the front overleaf of this application note.

iSBX™ MULTIMODULE™ BOARD CONCEPT

The iSBX MULTIMODULE board concept was developed to provide the users of Intel single board computers (iSBCs) with a convenient method to incrementally expand the I/O or the computing capabilities of a single board computer. This expansion is done through the use of a new interface called the iSBX bus interface. This interface gives the user the capability of adding I/O mapped functions directly onto the microprocessor bus via plug-in modules that connect to the iSBC board by means of a special iSBX connector. With the use of this new bus interface, it is now possible to expand or add new features to your iSBC system without incurring large costs and long engineering development times.

There are a number of unique advantages to using the iSBX bus interface for system expansion rather than adding a separate expansion board to your system. First, when expansion is required, the user needs only to buy what is required for the application. Second, it is now possible to return to one board solutions for small systems. One board solutions eliminate the need for expensive backplanes and cardcages. Next, the iSBX interface connects directly to the microprocessor or local bus, as opposed to interfacing to the MULTIBUS system bus, therefore I/O expansion does not require system bus cycles. To the CPU, the iSBX board looks like any other on-board I/O device (Figure 1). Address decode logic exists on the iSBC host board for each iSBX connector on the host board.

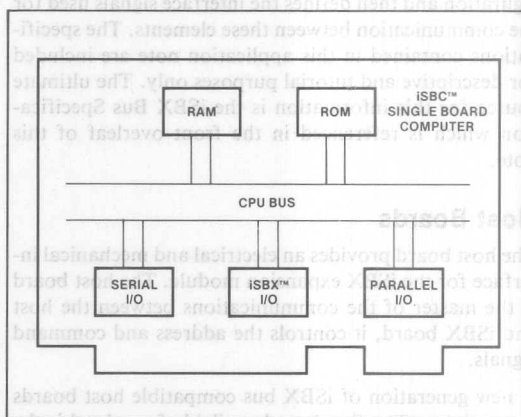


Figure 1. iSBC™ Host Board Block Diagram

Third, if there is no iSBC or MULTIBUS compatible expansion board available to fit the needs of your application or if the expansion boards available offer more capability than required, then it is possible to design a custom iSBX MULTIMODULE board. Custom iSBX boards offer several advantages over custom MULTIBUS boards: they require less board real estate (10 or 20

square inches versus 81 square inches) and less engineering design time; consequently, they cost considerably less to implement. Additional capability is therefore achieved with maximum productivity.

Currently available Intel iSBX MULTIMODULE Boards include:

- 1) iSBX 350 Parallel I/O MULTIMODULE board which contains 24 programmable I/O lines with sockets for line drivers and terminators.
- 2) iSBX 351 Serial I/O MULTIMODULE board containing one RS232 or RS449/422 programmable synchronous/asynchronous communications channel and two timers.
- 3) iSBX 331 Fixed/Floating Point Math MULTIMODULE board which permits fixed or floating point mathematics via the Intel 8231 device.
- 4) iSBX 332 Floating Point Math MULTIMODULE board which permits floating point mathematics using the Intel and proposed IEEE floating point standards via an Intel 8232 device.

With these iSBX MULTIMODULE boards and other soon-to-be-announced boards, the capability now exists to economically tailor a single board computer to the application using off-the-shelf products.

iSBX™ MULTIMODULE™ SYSTEM INTERFACE

This section begins by describing the basic system elements used in an iSBX MULTIMODULE interface configuration and then defines the interface signals used for the communication between these elements. The specifications contained in this application note are included for descriptive and tutorial purposes only. The ultimate source for this information is the iSBX Bus Specification which is referenced in the front-overleaf of this note.

Host Boards

The host board provides an electrical and mechanical interface for the iSBX expansion module. The host board is the master of the communications between the host and iSBX board, it controls the address and command signals.

A new generation of iSBX bus compatible host boards are evolving. The first board available from Intel is the iSBC 80/10B Single Board Computer. The 80/10B contains an 8080A CPU operating at 2 MHz, 1K bytes of RAM with sockets available for expansion to 4K bytes of RAM, sockets for up to 16K bytes of EPROM, 24 parallel I/O lines, a programmable synchronous/asynchronous communications interface and a fixed 1.04 msec timer. The 80/10B has one iSBX connector, permitting the use of an iSBX MULTIMODULE board.

The second iSBC board available supporting iSBX boards is the iSBC 80/24 Single Board Computer. The 80/24 board, which supports two iSBX MULTIMODULE boards, contains an 8085A-2 CPU operating at 4.8 or 2.4 MHz, 4K bytes of RAM, sockets for up to 32K bytes of EPROM, 48 parallel I/O lines, a programmable synchronous/asynchronous communications interface, three programmable interval timers and a programmable interrupt controller. Further RAM expansion on the 80/24 board is accomplished by the addition of an iSBC 301 4K byte RAM MULTIMODULE board which expands the RAM by an additional 4K bytes for a total of 8K bytes. The iSBC 301 MULTIMODULE board is not iSBX bus compatible; it is attached via pins and sockets in the RAM section of the host board.

iSBX™ MULTIMODULE™ Boards

The iSBX MULTIMODULE boards communicate with the host boards via the iSBX bus interface. These iSBX boards are I/O mapped through pre-defined select lines to specific port addresses. The iSBX bus currently defines an 8-bit data path compatible with both 8 and 16-bit future iSBC host boards. Examples of possible iSBX expansion boards include a floppy disk controller, a cassette interface, analog-to-digital converter or digital-to-analog converter boards, an interface to the IEEE 488 Bus and a video graphics display interface board.

There are two standard sizes of iSBX boards: a single-wide board measuring 7.24 by 9.40 cm (2.85 by 3.70 inches) and a double-wide board measuring 7.24 by 19.05 cm (2.85 by 7.50 inches). The iSBX MULTIMODULE boards mount onto any microcomputer board containing an iSBX connector and mounting hole. The iSBX boards physically plug into the iSBX connector on the host board and are secured with a nylon stand-off and screws. The mounting hardware supplied as part of the iSBX board includes:

- 1) One nylon spacer, 1/2" threaded
- 2) Two nylon screws, 1/4" 6-32
- 3) One 36-pin connector, factory-installed onto the iSBX module. (These may also be purchased from Intel.)

The interconnection between the host board and iSBX board, as well as the mounting clearances, may be seen in Figures 2 and 3.

NOTE

The iSBX board, when installed onto a host board, occupies an additional card slot adjacent to the base board in an iSBC 604/614 Cardcage. However, the base board may be inserted in the top card slot of the cardcage. If this is done, no additional slots are required.

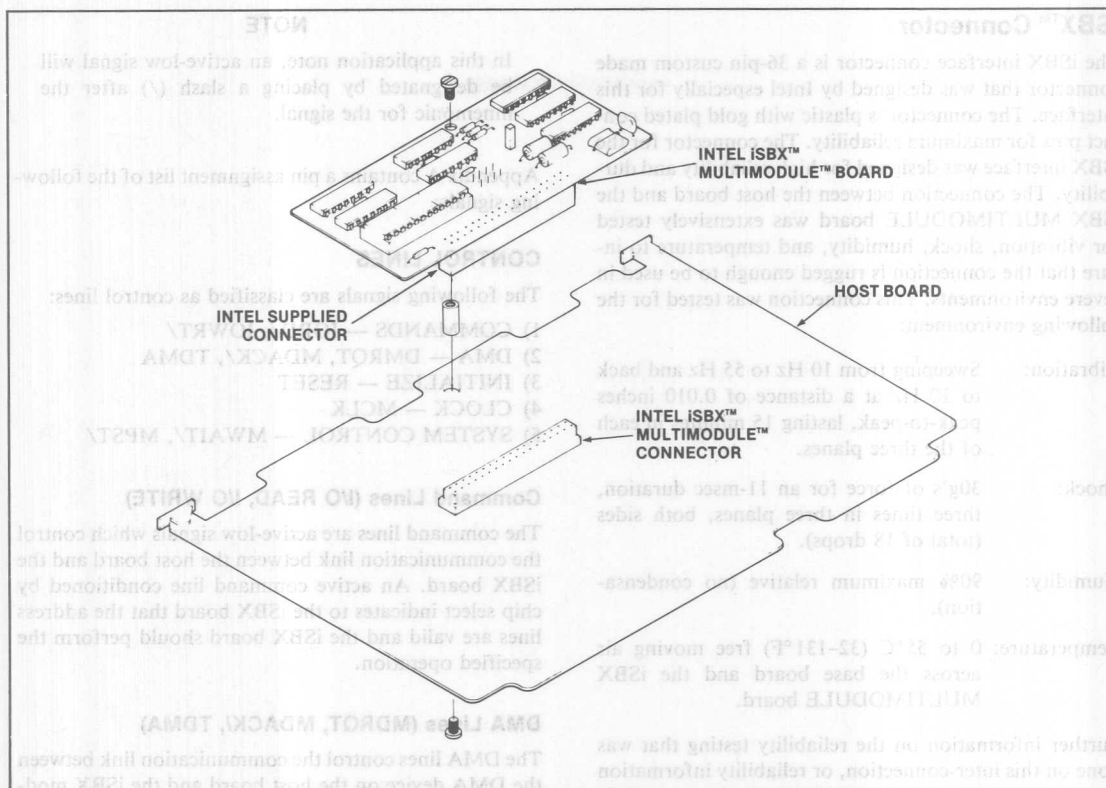


Figure 2. Connection of iSBX™ MULTIMODULE™ to Host Board

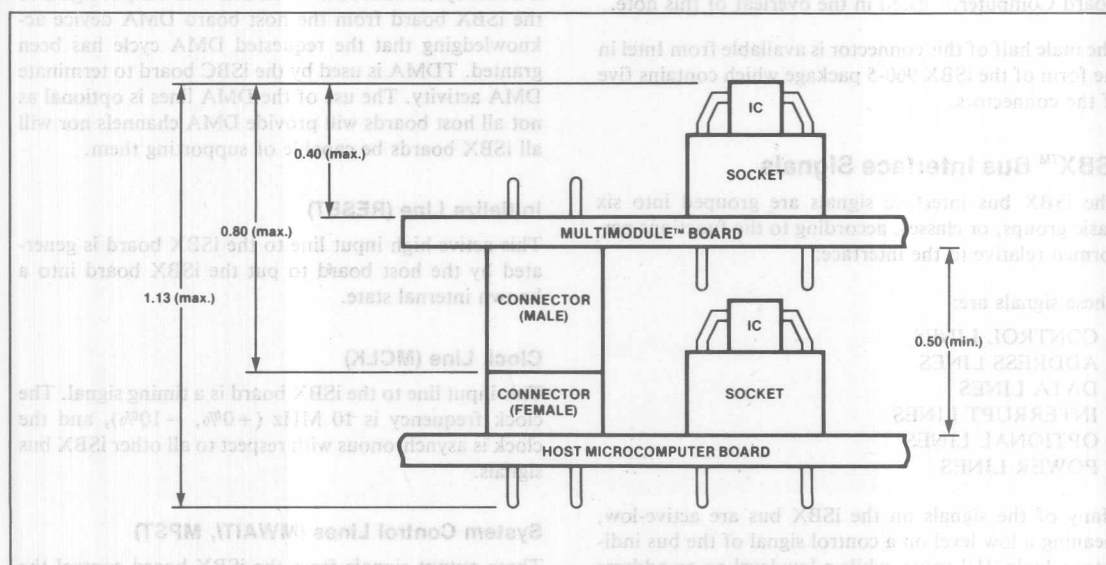


Figure 3. iSBX™/iSBC™ Mounting Clearance (inches)

iSBX™ Connector

The iSBX interface connector is a 36-pin custom made connector that was designed by Intel especially for this interface. The connector is plastic with gold plated contact pins for maximum reliability. The connector for the iSBX interface was designed for high reliability and durability. The connection between the host board and the iSBX MULTIMODULE board was extensively tested for vibration, shock, humidity, and temperature to insure that the connection is rugged enough to be used in severe environments. This connection was tested for the following environment:

- Vibration:** Sweeping from 10 Hz to 55 Hz and back to 10 Hz at a distance of 0.010 inches peak-to-peak, lasting 15 minutes in each of the three planes.
- Shock:** 30g's of force for an 11-msec duration, three times in three planes, both sides (total of 18 drops).
- Humidity:** 90% maximum relative (no condensation).
- Temperature:** 0 to 55°C (32-131°F) free moving air across the base board and the iSBX MULTIMODULE board.

Further information on the reliability testing that was done on this inter-connection, or reliability information on the iSBX MULTIMODULE boards in general, is contained in the Reliability Report, RR-29, "Intel iSBX MULTIMODULE Boards and iSBC 80/10B Single Board Computer," listed in the overleaf of this note.

The male half of this connector is available from Intel in the form of the iSBX 960-5 package which contains five of the connectors.

iSBX™ Bus Interface Signals

The iSBX bus interface signals are grouped into six basic groups, or classes, according to the functions performed relative to the interface:

These signals are:

- CONTROL LINES
- ADDRESS LINES
- DATA LINES
- INTERRUPT LINES
- OPTIONAL LINES
- POWER LINES

Many of the signals on the iSBX bus are active-low, meaning a low level on a control signal of the bus indicates a logic "1" value, while a low level on an address or data signal of the bus represents a logic "0" value.

NOTE

In this application note, an active-low signal will be designated by placing a slash (/) after the mnemonic for the signal.

Appendix A contains a pin assignment list of the following signals:

CONTROL LINES

The following signals are classified as control lines:

- 1) COMMANDS — IORD/, IOWRT/
- 2) DMA — DMRQT, MDACK/, TDMA
- 3) INITIALIZE — RESET
- 4) CLOCK — MCLK
- 5) SYSTEM CONTROL — MWAIT/, MPST/

Command Lines (I/O READ, I/O WRITE)

The command lines are active-low signals which control the communication link between the host board and the iSBX board. An active command line conditioned by chip select indicates to the iSBX board that the address lines are valid and the iSBX board should perform the specified operation.

DMA Lines (DMRQT, MDACK/, TDMA)

The DMA lines control the communication link between the DMA device on the host board and the iSBX module. DMRQT is an active-high output signal from the iSBX board to the host board's DMA device requesting a DMA cycle. MDACK/ is an active-low input signal to the iSBX board from the host board DMA device acknowledging that the requested DMA cycle has been granted. TDMA is used by the iSBC board to terminate DMA activity. The use of the DMA lines is optional as not all host boards will provide DMA channels nor will all iSBX boards be capable of supporting them.

Initialize Line (RESET)

This active-high input line to the iSBX board is generated by the host board to put the iSBX board into a known internal state.

Clock Line (MCLK)

This input line to the iSBX board is a timing signal. The clock frequency is 10 MHz (+0%, -10%), and the clock is asynchronous with respect to all other iSBX bus signals.

System Control Lines (MWAIT/, MPST/)

These output signals from the iSBX board control the state of the system. Active MWAIT/ (active-low) will

put the CPU on the host board into a wait state, providing additional time for the iSBX board to perform the requested operation. MPST/ is an active-low signal (usually tied to signal ground) that informs the host board I/O decode logic that an iSBX module has been installed.

ADDRESS AND CHIP SELECT LINES

The address and chip select lines are made up of the following signals:

- 1) ADDRESS LINES — MA0, MA1, MA2
- 2) CHIP SELECT LINES — MCS0/, MCS1/

Address Lines (MA0, MA1, MA2)

These active-high input lines to the iSBX boards are generally the least three significant bits of the I/O addresses. In conjunction with the command and chip select lines, they establish the I/O port address being accessed.

Chip Select Lines (MCS0/, MCS1/)

These active-low input lines to the iSBX board are the result of the host board I/O decode logic. When active, the MCS/ lines condition the I/O command signals and thus enable communication between the iSBX board and the host board.

DATA LINES (MD0-MD7)

There are eight bidirectional data lines. These active-high lines are used to transmit or receive information to or from the iSBX ports. MD0 is the least significant bit.

INTERRUPT LINES (MINTR0, MINTR1)

These active-high output lines from the iSBX board are used to make interrupt requests to the host board. These lines are jumper enabled and disabled on the host board via wire wrap posts.

OPTION LINES (OPT0, OPT1)

These two signals are reserved lines that are connected to wire wrap posts on both the host board and the iSBX MULTIMODULE board. They are for unique requirements where a user needs a host board or MULTIBUS bus signal on the iSBX module.

POWER LINES

All host boards provide +5 volts as well as ± 12 volts to the iSBX MULTIMODULE board along with signal ground. All power supply voltages are $\pm 5\%$. Table 1 gives the power supply specifications for the iSBX interface.

Table 1. Power Supply Specifications

Minimum (volts)	Nominal (volts)	Maximum (volts)	Maximum (current)*
+4.75	+5.0	+5.25	3.0A
+11.4	+12	+12.6	1.0A
-12.6	-12	-11.4	1.0A
—	GND	—	6.0A

*Per iSBX MULTIMODULE board mounted on base board.

iSBX™ BUS INTERFACING

This section of the application note focuses on the iSBX interface and design considerations related to interfacing with the iSBX bus. It discusses the way the major operations like READ, WRITE, and DMA work, and the timing diagrams associated with each. There is also a discussion on other considerations for designing with the iSBX bus.

Bus Timing

The AC timing specifications for the iSBX bus interface can be found in Appendix B of this application note. It should be emphasized that the interface timing between the host board and the iSBX MULTIMODULE board is very critical. This is largely due to the fact that the iSBX board is attached directly to the microprocessor bus. If the timing specifications are not met, unpredictable and possibly intermittent operation of the host board may result.

Command Operations

The command lines (IORD/, IOWRT) are driven from the host board by three-state drivers with pull-up resistors or standard TTL totem-pole drivers. These lines indicate to the iSBX board that action is being requested. There are two types of operations for each command line and it is the iSBX board that determines which operation is to be performed.

READ OPERATIONS (IORD/)

Two different types of read operations are possible. The first type of read is called a full speed I/O READ. The host board generates a valid I/O address (MA0-MA2) and a valid chip select signal (MCS1/) which is then sent to the iSBX board; after the set-up times are met, the host board activates the IORD/ line. At this time, the iSBX board must generate valid data from the addressed I/O port in less than 250 ns. The host board then reads the data and removes the READ command, address and chip selects. These are shown in the timing diagram for this operation (Figure 4). The second type of read operation is called an I/O READ with Wait. This READ is used by iSBX boards that cannot perform a full speed read operation. Under this operation the

host board generates the valid address and chip select signals, as in the full speed read. But this time the iSBX board will activate the MWAIT/ signal, which in turn removes the READY input to the CPU, putting it into a Wait state. The CPU, however, first activates the IORD/ signal before going into the Wait state. After valid data is placed on the iSBX data bus by the iSBX board, the iSBX board will remove the MWAIT/ signal. The host board will then read the data and remove the command, address, and chip select lines. This I/O READ with Wait operation is shown in Figure 5.

WRITE OPERATIONS (IOWRT/)

There are also two types of write operations possible: the type performed is again determined by the iSBX board. In the full speed I/O WRITE operation, the host board generates a valid I/O address and chip select and then activates the IOWRT/ line after the necessary setup times are met. The IOWRT/ line, after being activated, will remain active for 300 ns and the data will be valid for 250 ns before the IOWRT/ command is re-

moved. The host board will then remove the data, address, and chip select lines after the hold times are met, as shown in the timing diagram of this operation (Figure 6).

This second write operation is the I/O WRITE with Wait operation. This WRITE is used by the iSBX boards that cannot write into an I/O port with the full speed write specifications. The host board again generates valid address and chip select signals as in the full speed write operation. However, this time the iSBX board generates the MWAIT/ signal based on address information (chip select and MA0-MA1). The activation of MWAIT/ causes the removal of READY to the CPU, thus causing the CPU to go into a Wait state. The iSBX board removes the MWAIT/ signal (allowing the CPU to leave its Wait state) when it has satisfied the WRITE pulse width requirements. At this time the board removes the WRITE command, followed by the data, address, and chip select lines. This I/O WRITE with Wait operation can be seen in Figure 7.

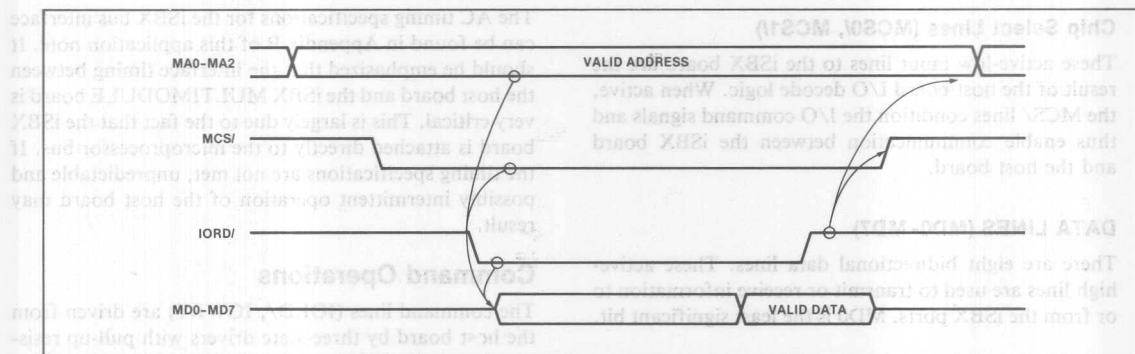


Figure 4. Full Speed I/O Read Operation

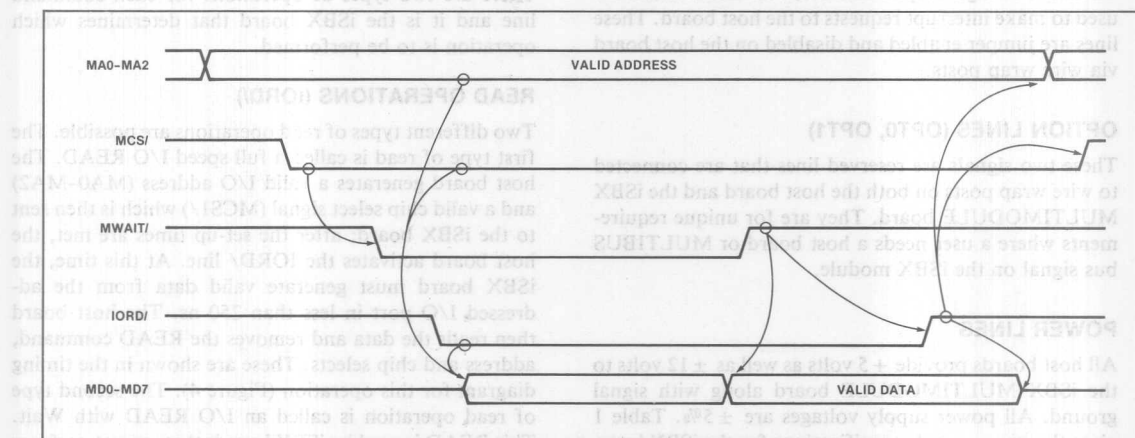


Figure 5. I/O Read with Wait Operation

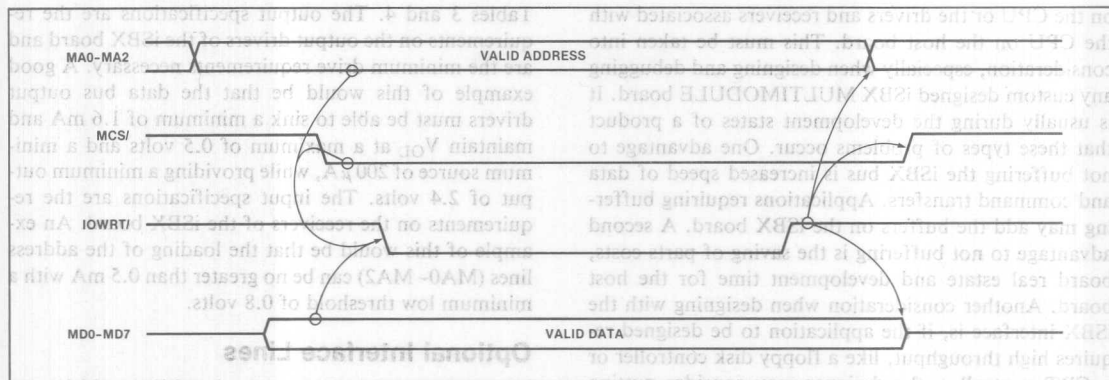


Figure 6. Full Speed I/O Write Operation

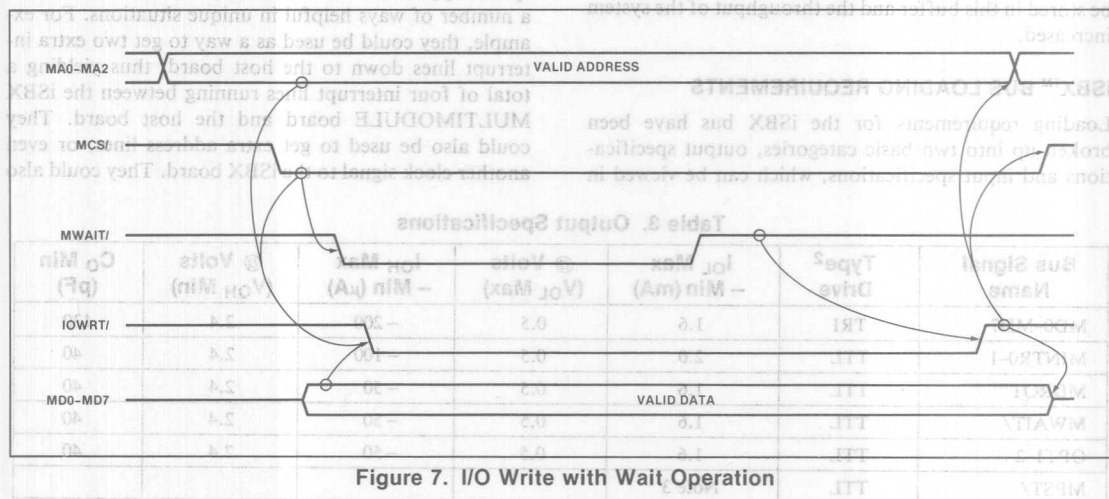


Figure 7. I/O Write with Wait Operation

iSBX™ Addressing

The iSBX boards are addressed by the host board through the use of the address lines MA0, MA1 and MA2, and the chip select lines MCS0/ and MCS1/. The host board decodes the I/O addresses and in turn generates the chip selects for the iSBX boards. In an 8-bit system the host board decodes the high order 13 address bits and generates the appropriate chip select corresponding to those address bits. The low order three address bits are passed to the iSBX board via MA0-MA2. Thus, a host board reserves two blocks of eight I/O ports for each iSBX connector. There can be as many as three iSBX connectors per host board, therefore a total of 48 addresses or six blocks of eight I/O ports that can be reserved for the iSBX boards. Table 2 contains a list of the I/O addresses and their corresponding host board iSBX port assignments of the iSBC 80/10B and iSBC 80/24 host boards.

Table 2. iSBX™ Host Board Port Assignment

iSBX™ Connector Number	Chip Select	iSBX™ Port Addresses
iSBC 80/10B Connector	MCS0/ MCS1/	F0-F7 F8-FF
iSBC 80/24 First Connector	MCS0/ MCS1/	F0-F7 F8-FF
iSBC 80/24 Second Connector	MCS0/ MCS1/	C0-C7 C8-CF

Considerations for iSBX™ Bus Interfacing

When designing with the iSBX interface it is important to note that the iSBX bus is not buffered on the host board. Since there is no isolation between the iSBX board and the host board CPU bus, a short between signal lines and power or ground could have a direct effect

on the CPU or the drivers and receivers associated with the CPU on the host board. This must be taken into consideration, especially when designing and debugging any custom designed iSBX MULTIMODULE board. It is usually during the development states of a product that these types of problems occur. One advantage to not buffering the iSBX bus is increased speed of data and command transfers. Applications requiring buffering may add the buffers on the iSBX board. A second advantage to not buffering is the saving of parts costs, board real estate and development time for the host board. Another consideration when designing with the iSBX interface is, if the application to be designed requires high throughput, like a floppy disk controller or a CRT controller, the designer may consider putting some type intelligent control of buffer RAM onto the iSBX board. By doing this, the transfer information can be stored in this buffer and the throughput of the system increased.

iSBX™ BUS LOADING REQUIREMENTS

Loading requirements for the iSBX bus have been broken up into two basic categories, output specifications and input specifications, which can be viewed in

Tables 3 and 4. The output specifications are the requirements on the output drivers of the iSBX board and are the minimum drive requirements necessary. A good example of this would be that the data bus output drivers must be able to sink a minimum of 1.6 mA and maintain V_{OL} at a maximum of 0.5 volts and a minimum source of $200 \mu A$, while providing a minimum output of 2.4 volts. The input specifications are the requirements on the receivers of the iSBX board. An example of this would be that the loading of the address lines (MA0- MA2) can be no greater than 0.5 mA with a minimum low threshold of 0.8 volts.

Optional Interface Lines

The iSBX interface has two optional lines which were included for the user to configure the iSBX board for special application needs. These two lines can be used in a number of ways helpful in unique situations. For example, they could be used as a way to get two extra interrupt lines down to the host board, thus yielding a total of four interrupt lines running between the iSBX MULTIMODULE board and the host board. They could also be used to get extra address lines, or even another clock signal to the iSBX board. They could also

Table 3. Output Specifications

Bus Signal Name	Type ² Drive	I _{OL} Max – Min (mA)	@ Volts (V _{OL} Max)	I _{OH} Max – Min (μA)	@ Volts (V _{OH} Min)	C _O Min (pF)
MD0-MD7	TRI	1.6	0.5	- 200	2.4	130
MINTR0-1	TTL	2.0	0.5	- 100	2.4	40
MDRQT	TTL	1.6	0.5	- 50	2.4	40
MWAIT/	TTL	1.6	0.5	- 50	2.4	40
OPT1-2	TTL	1.6	0.5	- 50	2.4	40
MPST/	TTL	Note 3				

Table 4. Input Specifications

Bus Signal Name	Type ² Receiver	I _{IL} Max (mA)	@ Volts (V _{IN} Max)	I _{IH} Max (μA)	@ Volts (V _{IN} Min)	C _I Max (pF)
MD0-MD7	TRI	- 0.5	0.4	70	2.4	40
MA0-MA2	TTL	- 0.5	0.4	70	2.4	40
MCS0/-MCS1/	TTL	- 4.0	0.4	100	2.4	40
MRESET	TTL	- 2.1	0.4	100	2.4	40
MDACK/	TTL	- 1.0	0.4	100	2.4	40
IORD/ IOWRT/	TTL	- 1.0	0.4	100	2.4	40
MCLK	TTL	2.4	0.4	100	2.4	40
OPT1-OPT2	TTL	2.0	0.4	100	2.4	40

NOTES:

1. Per iSBX MULTIMODULE board.
2. TTL = standard totem-pole output. TRI = three-state.
3. iSBX MULTIMODULE board must connect this signal to ground.

be used to send a special status line to or from the iSBX MULTIMODULE board.

iSBX™ MULTIMODULE™ DESIGN EXAMPLE

This section covers the description of a custom iSBX MULTIMODULE board which uses the Intel 8279 Programmable Keyboard/Display Controller. This iSBX board, when added to an iSBC host board, provides an interface to a keyboard and display. A description of the hardware design considerations for breadboarding the hardware is presented. Following this, a software exerciser, useful for debugging the board, is described. A listing for the exerciser is contained in Appendix C.

Since the iSBX MULTIMODULE board was designed using the Intel 8279 Programmable Keyboard/Display Controller, a brief description of the 8279 is presented. The 8279 is a general purpose programmable keyboard and display I/O controller which was designed for use with the Intel microprocessors. The keyboard portion of this device is capable of providing a scanned interface to a 64-contact key matrix. It is also possible to interface to an array of sensors or a strobed keyboard, such as those of the Hall Effect or the ferrite variety. The 8279 provides a variety of keyboard inputs (i.e., 2-key lockout and N-key rollover), and all key entries are debounced

and strobed into an 8-character FIFO. The display portion provides the user with a scanned display interface for LED, incandescent, and other popular display technologies. Both numeric and alphanumeric segment displays may be used, as well as simple indicators. The 8279 is used in this iSBX design example to provide an interface of 2-key lockout with key debounce to a 64-character keyboard, and an interface for a 16-character, 18-segment alphanumeric display.

iSBX™ MULTIMODULE™ Board Design

The iSBX board that was designed for this application note contains a total of three IC's, the keyboard/display controller, a flip-flop, and a 3-to-8-line decoder. Figure 8 contains a block diagram of the hardware used in this design example. Figure 9 contains a schematic for the portion of the design example resident on the custom iSBX board.

The design offers the user some flexibility as to the type of display or keyboard to be attached. For example, if the application design was defined to be for a 7-segment, 16-character display (as the 8279 is designed to drive), a 4-to-16-line decoder along with the display drivers could be added to the iSBX board. Another idea would be to include everything except the display drivers and the display on the iSBX board, and to put the dis-

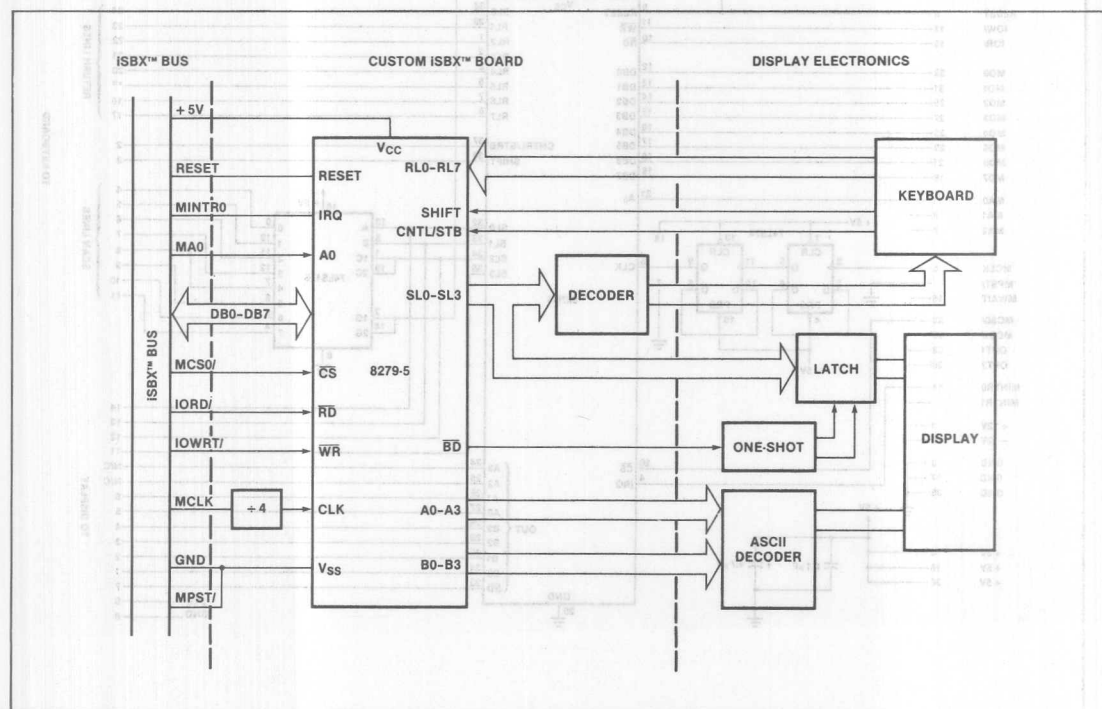


Figure 8. Block Diagram of the iSBX™ Design Example

play and drivers in with the keyboard. It is possible, and probably desirable in some applications, to incorporate some of the display electronics onto the iSBX MULTI-MODULE board. Some of the IC's found in the display portion of this design could also have been placed on the iSBX board, as there is enough room on the finished product for doing so.

The design was very easy to implement because, with the exception of one signal, all of the iSBX bus signals necessary to drive the 8279 are connected directly without any extra logic needed. The one signal that would not connect directly to the interface is the clock signal MCLK from the bus to CLK on the controller. It is not possible to connect these two together as MCLK is a 10 MHz signal and the 8279 requires a maximum clock signal of 3.1 MHz to generate its internal timings. It is necessary to add a 74LS74 dual D-type flip-flop to divide the MCLK signal by 4 for the controller. With this exception, all other signals, DB0-DB7 to MD0-MD7, A₀ to MA0, CS/ to MCS0/, etc., are connected directly

to the iSBX interface. To meet the timing requirements of the iSBX bus, a high speed version of the 8279, the 8279-5, is used.

The keyboard interface side of the iSBX board consists of a 3-to-8-line decoder, which is used for scanning the keyboard matrix. The 8279 scan lines SL0-SL2 are decoded by a 74LS156 open-collector output decoder and sent to the keyboard via a connector.

The display interface of the iSBX board consists of sending the scan lines and the display outputs to the display module via a connector. The scan lines SL0-SL3 are sent to the display drivers, and the display outputs A0-A3 and B0-B3 are sent to an ASCII to 18-segment decoder driver. The display is discussed in further detail in the next section of this application note.

Display Module Design

The display module design (Figure 10) consists of two 8-digit HDSP 6805 Alphanumeric Displays by Hewlett

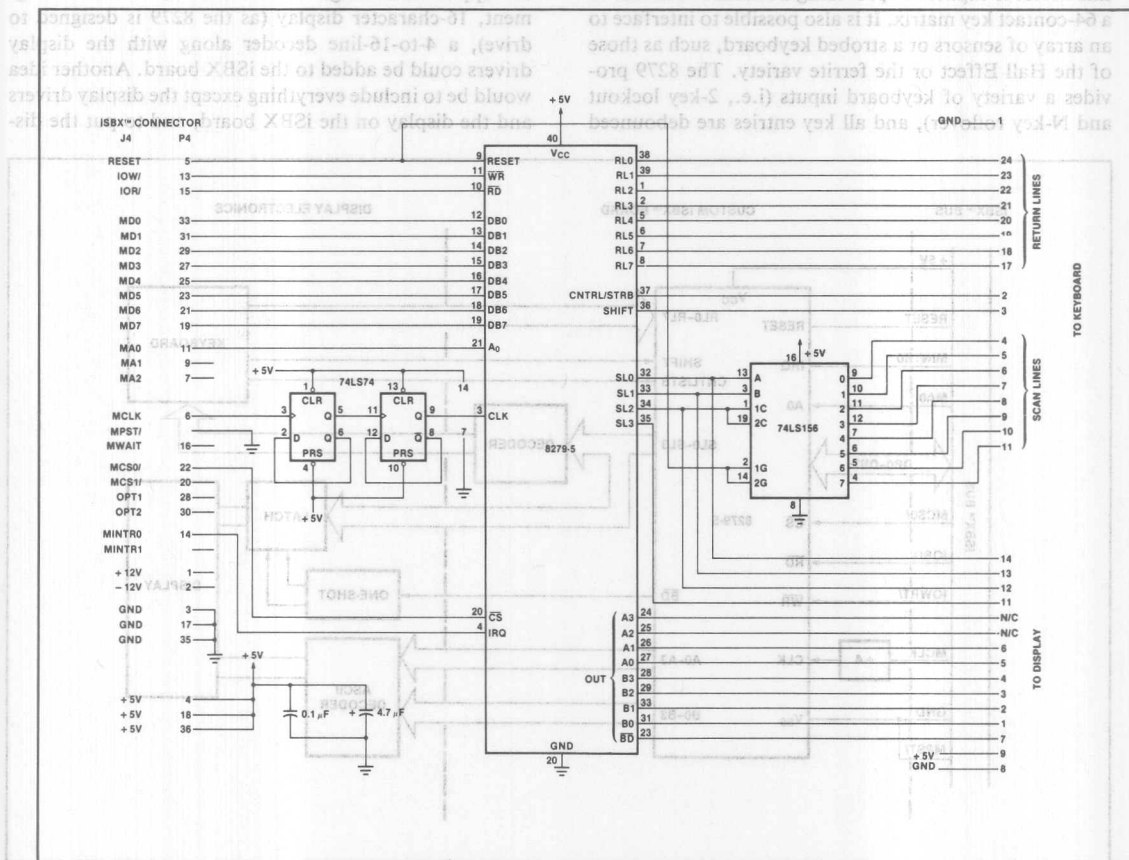


Figure 9. Schematic of the custom iSBX™ Board

Packard, the AC5947 ASCII to 18-segment decoder driver by Texas Instruments, two Signetics NE590 Peripheral Drivers, and a 74LS122 monostable multivibrator. The display is scanned by the outputs A0-A1 and B0-B3, which are connected to the inputs of the AC5947, and the SL0-SL3 outputs which are connected to the NE590 digit scanning circuitry. The interdigit blanking is provided by the 74LS122, which prevents a display ghosting type effect. With the 8279 display controller it is possible for the display to have either left entry, where the data enters from left to right across the display, overflowing in the left most display position, or right entry, where the data enters from the right side of the display and all previous data shifts left. Left entry was chosen for this example. The controller also provides commands for blanking or clearing the display.

Keyboard Interface Design

The eight output lines from the decoder on the iSBX board select 1-of-8 keyboard matrix rows for testing by the controller to see if a key depression has been made in the selected row. The keyboard matrix column output lines are connected directly to the return lines of the 8279, RL0-RL7. Open-collector outputs presented by individual keys within the matrix eliminate the need for isolation diodes when two keys in a given column are depressed. The keyboard/display controller has the option of using either scan keyboard, scan sensor matrix, or strobed input as modes of operation. With the scan keyboard mode there is a choice of using either 2-key lockout or N-key rollover for keyboard entry. The scan keyboard with 2-key lockout mode is used for this ex-

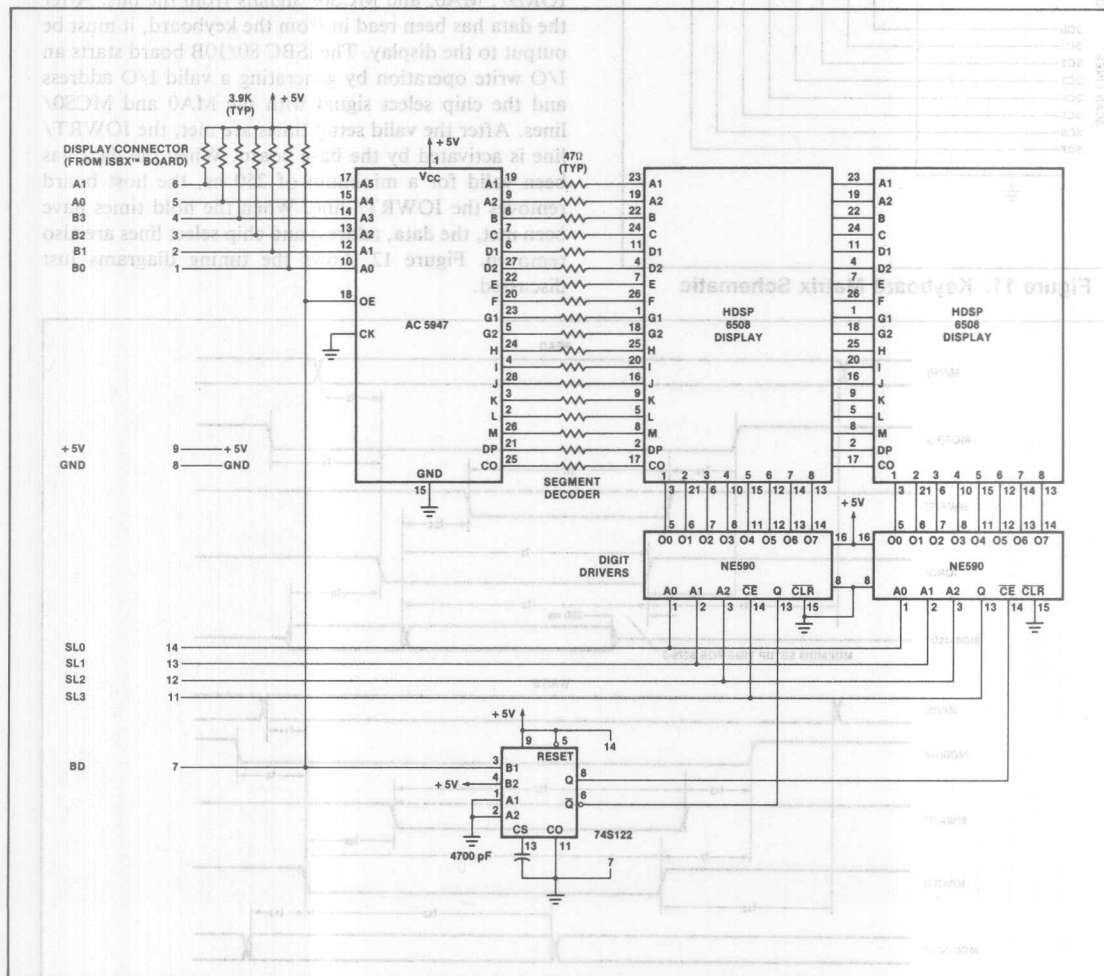


Figure 10. Display Module Schematic

ample. A diagram of the keyboard interfaces and matrix can be seen in Figure 11.

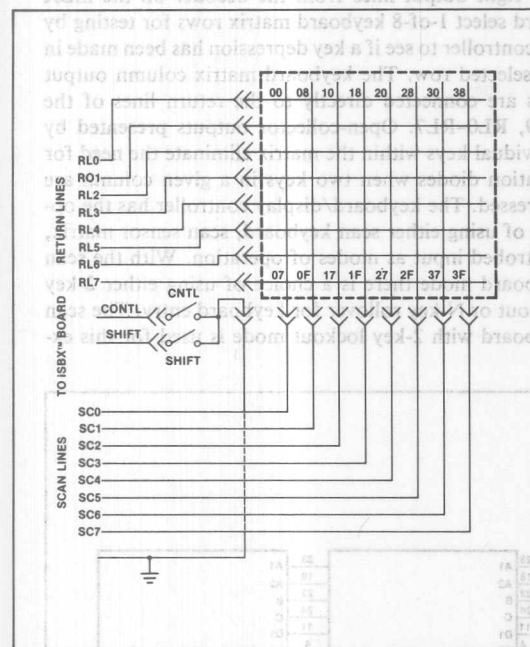


Figure 11. Keyboard Matrix Schematic

Operation with the iSBC 80/10B™ Single Board Computer

The 8279 on the iSBX expansion board is initialized to its mode of operation following a system reset. The keyboard mode of operation is to scan the keyboard with 2-key lockout, and the display mode is set for the 16-character left entry mode of operation. Upon receiving a character from the keyboard, the 8279 generates an interrupt along the MINTR0 line of the iSBX bus to the CPU. At this time the iSBC 80/10B board commences I/O read operations to the iSBX board by generating valid I/O address and chip select commands on the MA0 and MCS0/ signal lines. After the setup times are met, the 80/10B issues an I/O read command by asserting the IORD/ line on the bus, and the base board reads the data from the iSBX board and removes the IORD/, MA0, and MCS0/ signals from the bus. After the data has been read in from the keyboard, it must be output to the display. The iSBC 80/10B board starts an I/O write operation by generating a valid I/O address and the chip select signal with the MA0 and MCS0/ lines. After the valid setup times are met, the IOWRT/ line is activated by the base board. When the data has been valid for a minimum of 250 ns, the host board removes the IOWRT/ line. When the hold times have been met, the data, address and chip select lines are also removed. Figure 12 shows the timing diagrams just discussed.

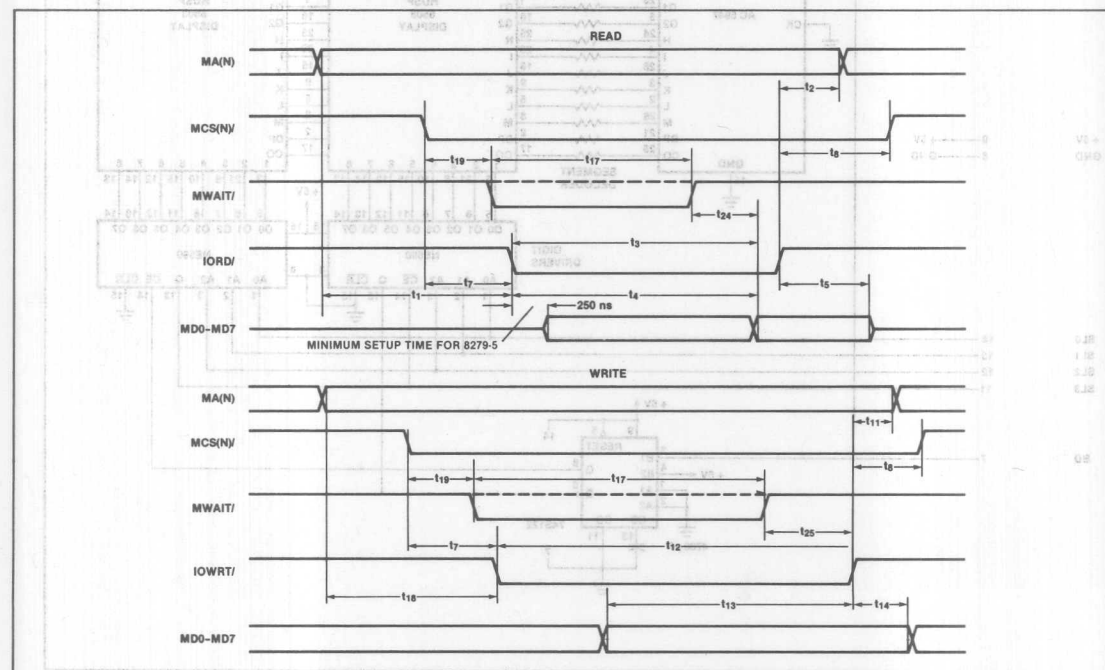


Figure 12. System Timing Diagrams

Breadboarding the Design

When doing the layout of the breadboard, it is also necessary to take into consideration the space required by the mounting holes and to plan the positioning of the components accordingly. (This information is available in the iSBX Bus Specification Manual.)

When attaching the breadboarded design, which typically contains raised wirewrap posts, it is necessary to raise the breadboard well above the host board. This can be accomplished by building a small cable and putting the breadboard on longer nylon standoffs. It is not recommended that the cable be longer than 15 cm (6 in.), otherwise bus timing problems could result.

With the breadboarding finished it is a good idea to re-check all wiring connections for possible errors. Also check all signal lines with an ohmmeter between power, and then ground, for potential shorts. An error at this point can cause serious damage to the host board!

Software Considerations

The software written for this application is an exerciser that is used for hardware checkout. It is a small program designed to echo characters from the keyboard to the display. The software was edited, assembled, linked and located with an Intel development system; it was then debugged with an in-circuit emulator. Both the software and the hardware debug is covered in the next section of this application note.

To facilitate this discussion the software exerciser is divided into three sections based upon the functions performed. The three functions are:

- 1) Keyboard interrupt routine
- 2) Initialization and flag checking routine
- 3) Character output routine

A complete listing of the software exerciser can be found in Appendix C.

KEYBOARD INTERRUPT ROUTINE

The 8279 generates an interrupt to the CPU whenever data is introduced into its FIFO/Sensor RAM. The interrupt is cleared by doing a data read. Whenever a key on the keyboard is depressed an interrupt is generated. Two things are required when an interrupt occurs. First, the keyboard input data must be retrieved and stored. Second, the interrupt routine must indicate that there is some data ready to be output to the display. Therefore, a buffer is created in memory (called "BUFF") at location 3C00H to store the keyboard data. A data present flag is set in a register (REG. C) to indicate that data is ready to be output and can be found in the buffer. In this way the interrupt routine is used to input characters from the keyboard to the input buffer. The buffer is then read by the output routine, which sends the characters to the display.

INITIALIZATION AND FLAG CHECKING ROUTINE

The initialization and flag checking routine first sets the stack pointer to the top of memory. After this the program proceeds to initialize the 8279 Keyboard/Display Controller to its proper mode of operation. The modes of operation used for this application note is scanned keyboard with 2-key lockout for the keyboard, and 16 characters with left entry for the display. As the 8279 has a desired internal operating frequency of 100 kHz, the frequency divider chain is programmed to divide by 19 hex, or 25 decimal. After the 8279 has been initialized, the program begins its next procedure of clearing the buffers. The keyboard input buffer, "BUFF", as well as the display buffer, "DBUFF", are both cleared to a blank display. This is done so that at the time of power up, the display will come up blank. With the initialization now complete, the program disables the interrupts and checks the data present flag for an indication that data might be present for output. If the data present flag is set, the output character routine is called; if it is not set, the interrupts are enabled and the program loops back around to check again. In summary, this routine initializes the 8279 and clears the buffers, and then loops on the data present flag looking for an indication that data is present in the input buffer. The input buffer is a one-byte wide buffer named "BUFF."

CHARACTER OUTPUT ROUTINE

The character output routine brings the character in from "BUFF" (the keyboard input buffer) and compares it to the characters located in a table. If the character can be matched to a character in the table it is replaced in "BUFF" with the corresponding character located in the same position of a second table. If there is no match, it is compared to the code for a control character. If there is no match with a control character, a compare is made to see if the character is a delete character. When a match is found and the acceptable character is placed in "BUFF", the output routine shifts the data in the display buffer (Figure 13) one position to the left and places the character from the input buffer into the display buffer at position "DBUFF" + 15. Now that

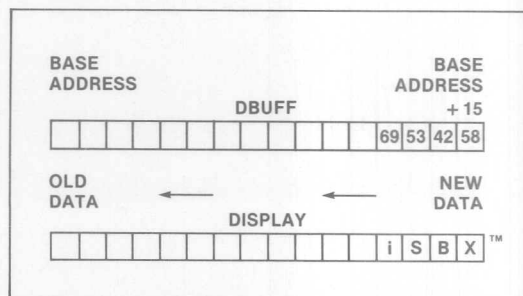


Figure 13. Display Buffer

the new information is in the display, the routine copies the complete contents of the display buffer, "DBUFF", to "DBUFF" + 15 to the display. In the case of the input character being matched up with a delete character, all information in the display buffer is shifted to the right one position and the ASCII code for a blank character is placed into the left-most position or the base address of "DBUFF", thus making the next character sent to the display a blank character. In the case of a control character, nothing is done and the program returns to the flag checking routine.

Debug Considerations

Hardware and software debug was accomplished using an iSBC 80/10B Single Board Computer, an iSBC 655 Chassis, an Intellec® Series II Model 230 Microcomputer Development System, and an ICE-80™ In-Circuit Emulator.

The software was down-loaded from the disk to the iSBC 80/10B board using the in-circuit emulator. The ICE™ module gives the engineer the capability of interrogating the iSBC system by allowing the user to access and display the CPU register contents, status, system memory contents, and all I/O devices and their data.

The iSBC 80/10B board was configured to enable interrupts from the iSBX board via the interrupt 0 line (MINTR0), which is connected to the interrupt pin of the 8080 CPU. The iSBX board was attached to the iSBC 80/10B board via the iSBX connector. The iSBC 80/10B board was powered-up and the iSBX board was

checked for proper power and ground connections. The ICE-80 emulator was connected to the iSBC 80/10B board. Using the interrogation mode of the emulator, it is possible to check proper functioning of the iSBX board by sending and receiving data to/from the 8279. The keyboard can be tested by depressing a key on the keyboard and then examining the FIFO/Sensor RAM to see if the data was entered. The display RAM can also be read and written to for testing the interface to the display.

After this initial checking of the iSBX board, the software exerciser can then be down-loaded with the ICE module to further check the board.

SUMMARY

The objective of this application note is to introduce the reader to the iSBX MULTIMODULE concept for expanding a single board computer's functionality, and to illustrate how a designer can use this concept with either standard or custom iSBX boards. In contrast to system expansion using MULTIBUS-compatible boards, iSBX MULTIMODULE boards provide smaller, lower cost, incremental expansion. This application note explains how a custom iSBX board can be designed and debugged. Using this capability, it is now possible to more quickly add new VLSI technology to systems as the technology becomes available. Intel will continue to provide new iSBX MULTIMODULE boards and, because of the the publication of the iSBX Bus Specification and this application note, it will be easier for Intel's customers to also design and build their own custom iSBX boards.

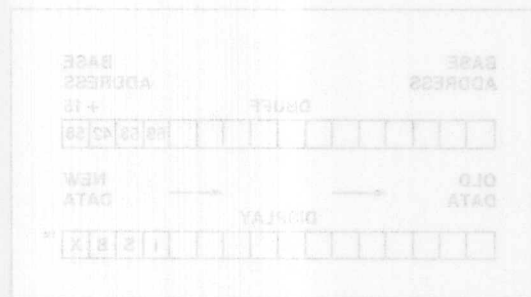


Figure 15. iSBX board

1) Keyboard interrupt routine
2) Initialization and flag checking routine
3) Character output routine
A complete listing of the software exerciser can be found in Appendix C.

KEYBOARD INTERRUPT ROUTINE

The CPU generates an interrupt to the CPU whenever the data is introduced into the FIFO/Sensor RAM. The interrupt is cleared by doing a data read. Whenever a key on the keyboard is depressed an interrupt is generated. Two things are required when an interrupt occurs. First, the keyboard input data must be retrieved and stored. Second, the interrupt routine must indicate that there is some data ready to be output to the display. Therefore, a buffer is created in memory (called "DBUFF") at location 1000H to store the keyboard data. A data present flag is set in a register (REG-C) to indicate that data is ready to be output and can be found in the buffer. In this way the interrupt routine is used to input characters from the keyboard to the input buffer. The buffer is then read by the output routine, which sends the characters to the display.

ISBX™ SIGNAL PIN ASSIGNMENTS APPENDIX A

Pin	Mnemonic	Description	Pin	Mnemonic	Description
32	GND	Signal Ground	36	+5V	+5 Volts
33	MD0	MDATA Bit 0	34	MDRQT	M DMA Request
34	MD1	MDATA Bit 1	35	MDACK	M DMA Acknowledge
35	MD2	MDATA Bit 2	36	OPT0	Option 0
36	MD3	MDATA Bit 3	37	OPT1	Option 1
37	MD4	MDATA Bit 4	38	TDMA	Terminate DMA
38	MD5	MDATA Bit 5	39	Reserved	Reserved
39	MD6	MDATA Bit 6	40	MCSTV	M Chip Select 0
40	MD7	MDATA Bit 7	41	Reserved	Reserved
41	GND	Signal Ground	42	Reserved	Reserved
42	IORDV	I/O Address Command	43	Reserved	Reserved
43	IOWRTV	I/O Write Command	44	Reserved	Reserved
44	MA0	M Address 0	45	Reserved	Reserved
45	MA1	M Address 1	46	Reserved	Reserved
46	MA2	M Address 2	47	MBSTV	ISBX MULTIMODULE Board Present
47	RESET	Reset	48	MCLK	M Clock
48	GND	Signal Ground	49	+5V	+5 Volts
49	+12V	+12 Volts	50	-12V	-12 Volts

All unassigned pins are reserved for future use.

APPENDIX A	1-192
APPENDIX B	1-193
APPENDIX C	1-194

APPENDIX A

iSBX™ SIGNAL PIN ASSIGNMENTS

Pin	Mnemonic	Description	Pin	Mnemonic	Description
35	GND	Signal Ground	36	+ 5V	+ 5 Volts
33	MD0	MDATA Bit 0	34	MDRQT	M DMA Request
31	MD1	MDATA Bit 1	32	MDACK/	M DMA Acknowledge
29	MD2	MDATA Bit 2	30	OPT0	Option 0
27	MD3	MDATA Bit 3	28	OPT1	Option 1
25	MD4	MDATA Bit 4	26	TDMA	Terminate DMA
23	MD5	MDATA Bit 5	24		Reserved
21	MD6	MDATA Bit 6	22	MCS0/	M Chip Select 0
19	MD7	MDATA Bit 7	20	MCS1/	M Chip Select 1
17	GND	Signal Ground	18	+ 5V	+ 5 Volts
15	IORD/	I/O Read Command	16	MWAIT/	M Wait
13	IOWRT/	I/O Write Command	14	MINTR0	M Interrupt 0
11	MA0	M Address 0	12	MINTR1	M Interrupt 1
9	MA1	M Address 1	10		Reserved
7	MA2	M Address 2	8	MPST/	iSBX MULTIMODULE Board Present
5	RESET	Reset	6	MCLK	M Clock
3	GND	Signal Ground	4	+ 5V	+ 5 Volts
1	+ 12V	+ 12 Volts	2	- 12V	- 12 Volts

All undefined pins are reserved for future use.

APPENDIX B

iSBX™ MULTIMODULE™ BOARD I/O AC SPECIFICATIONS

Symbol*	Parameter	Min (ns)	Max (ns)
t ₁	Address stable before read	50	—
t ₂	Address stable after read	30	—
t ₃	Read pulse width	300	—
t ₄ ⁽²⁾	Data valid from read	0	250
t ₅ ⁽²⁾	Data float after read	0	150
t ₆	Time between RD and/or WRT	—	Note 3
t ₇	CS stable before CMD	25	—
t ₈	CS stable after CMD	30	—
t ₉	Power up reset pulse width	50 ms	—
t ₁₀	Address stable before WRT	50	—
t ₁₁	Address stable after WRT	30	—
t ₁₂ ⁽²⁾	Write pulse width	300	—
t ₁₃ ⁽²⁾	Data valid to write	250	—
t ₁₄	Data valid after write	30	—
t ₁₅	MCLK cycle	100	110
t ₁₆	MCLK width	35	65
t ₁₇ ⁽¹⁾	MWAIT/ pulse width	0	4 ms
t ₁₈	Reset pulse width	50 ms	—
t ₁₉	MCS/ to MWAIT/ valid	0	75
t ₂₀	DACK set up to I/O CMD	100	—
t ₂₁	DACK hold	30	—
t ₂₂	CMD to DMA RQT removed to end of DMA cycle	—	200
t ₂₃	TDMA pulse width	500	—
t ₂₄ ⁽¹⁾	MWAIT/ to valid read data	—	0
t ₂₅ ⁽¹⁾	MWAIT/ to WRT CMD	0	—

NOTES:

1. Required only if WAIT is activated.
 2. If MWAIT/ not activated.
 3. To be specified by each iSBX MULTIMODULE board.
- * For a more complete definition of symbols refer to iSBX Bus Specification, 142686-001.

APPENDIX C

LISTING FOR THE ISBX™ DESIGN EXAMPLE SOFTWARE EXERCISER

LOC	OBJ	LINE	SOURCE STATEMENT	Symbol
		1	*****	
		2	*****	
		3	THIS PROGRAM WAS USED AS AN EXAMPLE FOR EXERCISING THE	
		4	8279 ISBX MULTIMODULE BUILT FOR THIS APPLICATION NOTE.	
		5	*****	
		6	*****	
		7		
		8		
		9	*****	
		10	PROGRAM EQUATES	
		11	*****	
		12		
00F0		13	DATAAD EQU 0F0H ; PORT ADDRESS TO READ OR WRITE	
		14	;/DATA TO/OR FROM KEYBOARD/DISPLAY	
00F1		15	CMDAD EQU 0F1H ; PORT ADDRESS TO SEND COMMANDS	
		16	;/TO KEYBOARD/DISPLAY	
0008		17	MODE0 EQU 08H ; CONTROL CHAR. TO SET	
		18	;/KEYBOARD/DISPLAY MODE FOR	
		19	;/C2 KEY LOCKOUT,16 CHAR LEFT ENTRY	
0039		20	PROGCK EQU 39H ; CONTROL CHAR. TO SET 8279 CLK	
		21	;/TO 100 KHZ INTERNAL TIMING	
0040		22	RDFIFO EQU 40H ; CONTROL CHAR. TO READ KEYBOARD	
0060		23	RDRAM EQU 60H ; CONTROL CHAR. TO READ DISPLAY RAM	
0070		24	RDRAMA EQU 70H ; CONTROL CHAR. TO READ DISPLAY RAM	
		25	;/AUTO INCREMENT	
0080		26	WRRAM EQU 80H ; CONTL CHAR. TO WRITE TO DISPLAY RAM	
0090		27	WRRAMA EQU 90H ; CONTL CHAR. TO WRITE TO DISPLAY	
		28	;/RAM AUTO INCREMENT	
00D8		29	CLR EQU 0D8H ; CONTROL CHAR. TO CLEAR OR BLANK	
		30	;/DISPLAY	
3C00		31	BUFF EQU 3C00H ; ADDRESS OF KEYBOARD INPUT BUFFER	
3D00		32	DBUFF EQU 3D00H ; ADDRESS OF DISPLAY BUFFER	
		33		
		34	*****	
		35		
0000 F3		36	START: DI BEGIN	
0001 C33B00		37	JMP	
		38		
		39	***** RST 7 ENTRY POINT *****	
		40		
0038		41	ORG 38H	
0038 C3D100		42	JMP INT	
		43		
		44	*****	
		45	INITIALIZE PROGRAM	
		46	AND KEY BOARD DISPLAY CONTROLLER	
		47		
003B 31FF3F		48	BEGIN: LXI SP,3FFFH ; INITIALIZE STACK PT	
003E 3E08		49	MVI A,MODE0 ; GET CONTROL CHAR.	
0040 D3F1		50	OUT CMDAD ; SET KEYBOARD/DISPLAY MODE	
0042 3E39		51	MVI A,PROGCK ; GET CONTROL CHAR.	
0044 D3F1		52	OUT CMDAD ; SET 8279 CLK FOR 100 KHZ	
0046 3E08		53	MVI A,CLR ; GET CONTROL CHAR.	
0048 D3F1		54	OUT CMDAD ; CLEAR OR BLANK DISPLAY	
004A 0EE0		55	MVI C,0E0H ;	
004C 21003C		56	LXI H,BUFF ; SET POINTER TO INPUT BUFFER	
004F 71		57	MOV M,C ; CLEAR INPUT BUFFER TO BLANK CODE	
0050 060F		58	MVI B,0FH ; SET COUNTER = 15	
0052 210F3D		59	LXI H,DBUFF+0FH ; SET POINTER TO DBUFF +15	
0055 71		60	MOV M,C ; CLEAR DISPLAY BUFFER TO	
0056 2B		61	DCX H ; /DISPLAY BUFFER +15 TO CODE	
0057 05		62	DCR B ; /FOR CLEARING OR BLANKING OUT	
0058 C25500		63	JNZ DBUFF ; /THE DISPLAY	
		64		
		65	*****	
		66	THIS IS THE BACKGROUND PROGRAM	
		67	WHICH LOOPS CHECKING FOR THE DATA PRESENT FLAG	
		68		
005B F3		69	CKFLAG: DI ; DISABLE INTERRUPTS	
005C AF		70	XRA A ; /CLEAR A REG AND COMPARE WITH	
005D B9		71	CMP C ; /C REG CHECKING FOR DATA PRESENT	
005E CA6400		72	JZ LABEL ; /IF PRESENT CALL OUTPT	
0061 CD6800		73	CALL OUTPT ; /TO DISPLAY CHAR.	
0064 FB		74	LABEL: EI ; /IF NO DATA PRESENT ENABLE	
0065 C35B00		75	JMP CKFLAG ; /INTERRUPTS AND JMP BACK	
		76		

```

***** 77 ;*****
78 ;          OUTPUT CHARACTER TO DISPLAY
79 ;*****
0068 3A003C 80 OUTPT: LDA BUFF ; LOAD A WITH KEYBOARD DATA
006B 062B 81 MVI B,2BH ; SET COUNTER MAX POSSIBLE CHAR.
006D 21DE00 82 LXI H,TABLE1 ; SET POINTER TO INPUT TABLE
0070 110901 83 LXI D,TABLE2 ; SET POINTER TO OUTPUT TABLE
0073 BE 84 COMPARE: CMP M ; COMPARE KEYBD DATA TO INPUT
0074 CA8000 85 JZ MATCH ;/TABLE IF = JMP TO MATCH
0077 05 86 DCR B ;/ELSE DECREMENT COUNTER IF 0
0078 CAC600 87 JZ CONTROL ;/JMP TO CONTROL
007B 23 88 INX H ;/ELSE INCREMENT BOTH TABLE
007C 13 89 INX D ;/POINTERS AND JMP TO COMPARE
007D C37300 90 JMP COMPARE
0080 EB 91 MATCH: XCHG ; IF MATCH CHANGE INPUT WITH
0081 7E 92 MOV A,M ;/OUTPT DATA AND PLACE IN BUFF
0082 21003C 93 LXI H,BUFF
0085 77 94 MOV M,A
0086 060F 95 MVI B,0FH ; SET COUNTER = TO 15
0088 11003D 96 LXI D,DBUFF ; POINTER TO FIRST LOC IN DBUFF
008B 21013D 97 LXI H,DBUFF+1 ; POINTER TO 2ND LOC IN DBUFF
008E 7E 98 LOOP1: MOV A,M ; READ HIGH POINTER FROM DBUFF
008F 23 99 INX H ;/UPDATE HIGH POINTER
0090 EB 100 XCHG
0091 77 101 MOV M,A ; SHIFT DATA LEFT IN D BUFF
0092 23 102 INX H ; UPDATE LOW POINTER
0093 EB 103 XCHG
0094 05 104 DCR B ; TEST IF DONE
0095 C28E00 105 JNZ LOOP1 ;/AND GO BACK IF NOT
0098 3A003C 106 LDA BUFF ;/ELSE READ KEYBOARD DATA
009B 320F3D 107 STA DBUFF+0FH ;/AND PLACE IT IN THE DBUFF
009E 0610 108 LOOPA: MVI B,10H ; SET COUNTER = 16
00A0 21003D 109 LXI H,DBUFF ; SET POINTER = DBUFF 1ST POS.
00A3 7E 110 LOOP2: MOV A,M ;/READ 1 BYTE FROM DBUFF
00A4 D3F0 111 OUT DATAAD ;/AND SENT IT TO DISPLAY
00A6 23 112 INX H ; UPDATE POINTER
00A7 05 113 DCR B ;/AND TEST IF DONE
00A8 C2A300 114 JNZ LOOP2 ;/GO BACK IF NOT DONE
00AB 0E00 115 MVI C,0H ;/ELSE CLR DATA PRESENT FLAG
00AD C9 116 RET ;/AND RETURN
117 ;*****
118 ;          CHARACTER DELETE
119 ;          OR RUB OUT
120 ;
00AE 060F 121 DELETE: MVI B,0FH ; SET COUNTER =15
00B0 110F3D 122 LXI D,DBUFF+0FH ; SET POINTER = DBUFF+15
00B3 210E3D 123 LXI H,DBUFF+0EH ; SET POINTER = DBUFF+14
00B6 7E 124 LOOPB: MOV A,M ; READ LOW POINTER FROM DBUFF
00B7 2B 125 DCX H ;/UPDATE LOW POINTER
00B8 EB 126 XCHG
00B9 77 127 MOV M,A ; SHIFT DATA RIGHT IN DBUFF
00BA 2B 128 DCX H ;/UPDATE HIGH POINTER
00BB EB 129 XCHG
00BC 05 130 DCR B ; TEST IF DONE
00BD C2B600 131 JNZ LOOPB ;/AND GO BACK IF NOT
00C0 EB 132 XCHG ;/ELSE SET DBUFF FOR
00C1 36E0 133 MVI M,0E0H ;/CODE TO BLANK DISPLAY
00C3 C39E00 134 JMP LOOPA ;/AND JMP TO LOOPA
135 ;
136 ;*****
137 ;          CHECK IF CHARACTER IS
138 ;          A CONTROL OR DELETE CHARACTER
139 ;
00C6 FEFA 140 CONTROL: CPI 0FAH ; COMPARE FOR CONTROL CHAR.
00C8 CA3B00 141 JZ BEGIN ;/IF CONTROL JMP TO BEGIN
00CB FEF9 142 CPI 0F9H ;/ELSE COMP. FOR DELETE CHAR.
00CD CAAE00 143 JZ DELETE ;/IF DELETE JMP TO DELETE
00D0 C9 144 RET ;/ELSE RETURN
145 ;
146 ;*****
147 ;          KEYBOARD INPUT
148 ;          INTERRUPT ROUTINE
149 ;
00D1 3E40 150 INT: MVI A,RDFIFO ; GET CONTL CHAR. TO READ FIFO
00D3 D3F1 151 OUT CNDAD ; SET 8279 FOR READ MODE
00D5 DBF0 152 IN DATAAD ; READ KEYBOARD DATA IN
00D7 21003C 153 LXI H,BUFF ; SET POINTER TO BUFF
00DA 77 154 MOV M,A ;/AND STORE KEYBOARD DATA
00DB 0EFF 155 MVI C,0FFH ;/THEN SET DATA PRESENT FLAG
00DD C9 156 RET ;/AND RETURN
157 ;

```

	158 ;	*****
	159 ;	TABLE 1
	160 ;	ACCEPTABLE INPUT CHARACTERS FROM KEYBOARD
	161 ;	*****
00DE DE	162 TABLE1: DB	0DEH,OFFH,0EFH,0EEH,0E5H,0F6H,0FFH,0C6H
00DF FF		
00EO EF		
00EI EE		
00E2 E5		
00E3 F6		
00E4 FE		
00E5 C6		
00E6 C9	163 ***** DB	0C9H,0CAH,0D2H,0DAH,0D3H,0C7H,0D1H,0D9H
00E7 CA		
00E8 D2		
00E9 DA		
00EA D3		
00EB C7		
00EC D1		
00ED D9		
00EE D5	164 ***** DB	0D5H,0EDH,0E6H,0F5H,0C1H,0F7H,0DDH,0E7H
00EF DD		
00FO E6		
00F1 F5		
00F2 C1		
00F3 FD		
00F4 DD		
00F5 E7		
00F6 FD	165 ***** DB	0FDH,0DFH,0CCH,0D4H,0DCH,0E4H,0ECH,0F4H
00F7 FD		
00F8 CC		
00F9 D4		
00FA DC		
00FB E4		
00FC EC		
00FD FA		
00FE FC	166 ***** DB	0FCH,0COH,0CBH,0DOH,09BH,0A2H,0CFH,0AAH
00FF CO		
0100 C8		
0101 D0		
0102 98		
0103 A2		
0104 CF		
0105 AA		
0106 EB	167 ***** DB	0EBH,0E3H,0D8H
0107 E3		
0108 D8	168 ***** DB	

```

169 ;*****
170 ;          TABLE 2
171 ;          ACCEPTABLE OUTPUT CHARACTERS TO DISPLAY
172 ;
173 TABLE2:      DB          0C1H,0C2H,0C3H,0C4H,0C5H,0C6H,0C7H,0C8H

0109 C1
010A C2
010B C3
010C C4
010D C5
010E C6
010F C7
0110 C8
0111 C9
0112 CA
0113 CB
0114 CC
0115 CD
0116 CE
0117 CF
0118 D0
0119 D1
011A D2
011B D3
011C D4
011D D5
011E D6
011F D7
0120 D8
0121 D9
0122 DA
0123 F1
0124 F2
0125 F3
0126 F4
0127 F5
0128 F6
0129 F7
012A F8
012B F9
012C F0
012D FD
012E EB
012F E0
0130 EA
0131 EF
0132 EE
0133 2D
0000

174          DB          0C9H,0CAH,0CBH,0CCH,0CDH,0CEH,0CFH,0D0H

175          DB          0D1H,0D2H,0D3H,0D4H,0D5H,0D6H,0D7H,0D8H

176          DB          0D9H,0DAH,0F1H,0F2H,0F3H,0F4H,0F5H,0F6H

177          DB          0F7H,0F8H,0F9H,0F0H,0FDH,0EBH,0E0H,0EAH

178          DB          0EFH,0EEH,02DH

179          END          START

PUBLIC SYMBOLS

EXTERNAL SYMBOLS

USER SYMBOLS
BEGIN A 003B      BUFF A 3C00      CKFLAG A 005B      CLR A 00D8      CMDAD A 00F1      COMPAR A 0073      CONTRO A 00C6
DATAAD A 00F0     DBUFF A 3D00     DELETE A 00AE     INT A 00D1     LABEL A 0064     LOOP1 A 008E     LOOP2 A 00A3
LOOPA A 009E      LOOPB A 00B6      MATCH A 0080     MODE0 A 0008     OUTPT A 0068     PROGCK A 0039     RDFIFO A 0040
RDRAM A 0060      RDRAMA A 0070      START A 0000     TABLE1 A 00DE  TABLE2 A 0109     WRRAM A 0080     WRRAMA A 0090
ZDBUFF A 0055

ASSEMBLY COMPLETE, NO ERRORS

```


Reduce your uC-based system design time by using single-board microcomputers. Assembled boards in the SBC-80 series offer stock answers to custom demands.

The 8080A CPU, the 8254 clock generator and the 8255 system controller. Capable of fetching and executing any of the 8080A's 78 instructions, the CPU section can respond to interrupt requests originating on and off the board. (For more about the 8080A, see "Micro-

processor Basics, Part 2," ED No. 10, May 10, 1977, p. 84).

The system-bus interface section includes an assortment of circuits to gate the interrupt and hold requests, the ready signals, and a system-reset signal. Other circuits drive the various control lines. Two 8216s help drive the bidirectional data bus, and six 8235s drive the external system-data and address buses as part of the SBC-80V10A's Multiplex interface.

The RAM section of the 80V10A consists of 1024 bytes of static MOS memory. For program storage, up to 8192 bytes of ROM can be mounted on the board in 1024-byte increments by means of a 2708 or 2709 EPROM, an 8208 mask-programmed ROM, or in 2048 EPROMs via the 2716 EPROM or 2816 ROM.

A serial interface on the board uses an 8251 programmable universal synchronous/asynchronous receiver/transmitter to provide a serial-data channel. The serial port operates at programmable rates up to 38,400 baud (synchronously) or 18,200 baud (asynchronously) with a choice of character length, number of stop bits, and even, odd or no parity. On-board interfaces provide direct EIA RS-232C or teletypewriter current-loop compatibility.

The 8255 programmable peripheral interface provides 48 I/O lines for transferring data to or from peripheral devices. Eight already-committed lines have bidirectional drivers and termination networks permanently installed, so that they can be used for unidirectional (dumper-selector) or bidirectional (lumper-selector) data transfer. Other lines are uncommitted. On-board sockets are provided for expansion. The software configures the I/O lines. It can be used for every application. The 80V10A provides a single-level interrupt that originates from many sources, the user-designated I/O interrupt is recognized. When an interrupt is recognized, the processor location is noted and the address of the first instruction to be executed is stored. System expansion and support are possible with a

System designers eager to take advantage of the dramatically increased capabilities of microcomputers have been hindered two ways: Their production volumes have been too low to amortize software development costs effectively, or

hardware suppliers and test laboratories have confined them to fully assembled and tested computer subsystems. But now those obstacles are overcome with families of fully assembled and tested microcomputers and system-expansion boards like the Intel SBC-80 series. They are ready-to-use, flexible and inexpensive—prices range from just \$185 to \$235 in unit quantities.

The main members of the SBC-80 family are the 80V10A, 80V20, 80V30 and 80V30-4 central-processor boards. Each contains either an 8080A or 8085 microprocessor acting as master CPU (Table I). Most of the boards incorporate 12 in. and contain the CPU, clock, control ROM, control RAM, I/O ports, serial control interface and bus-control logic.

I/O interfacing and bus-control flexibility is essential to meet changing requirements. The programmable serial interface structures of the boards make it very easy to test that. What's more, the SBC-80 family offers easy changes to the SBC-80V10A, which permits modular expansion. The Multiplex interface provides a standard interface between the SBC-80 single-board microcomputers and expansion boards. As many as 16 expansion boards can simultaneously share the bus.

All in the SBC-80 family

As exemplified by the block diagram in Figure 1, the SBC-80 microcomputers can handle all that's needed for many applications. The SBC-80V10A is the oldest board in the family and has been widely imitated since it was one of the first "standardized" microcomputers commercially available.

The CPU section of the 80V10A consists of the 8080A CPU, the 8254 clock generator and the 8255 system controller. Capable of fetching and executing any of the 8080A's 78 instructions, the CPU section can respond to interrupt requests originating on and off the board. (For more about the 8080A, see "Micro-

processor Basics, Part 2," ED No. 10, May 10, 1977, p. 84).

February 1, 1978

Reduce your uC-based system design time by using single-board microcomputers

By George Adams
Electronic Design 3/February 1, 1978

Reduce your μ C-based system design time by using single-board microcomputers. Assembled boards in the SBC-80 series offer stock answers to custom demands.

System designers eager to take advantage of the dramatically increased capabilities of microcomputers have been hindered two ways: Their production volumes have been too low to amortize software and hardware development costs effectively, or hardware subtleties and test requirements have confined them to fully assembled and tested computer subsystems. But now those obstacles are overcome with families of fully assembled and tested microcomputers and system-expansion boards like the Intel SBC-80 series. They are ready-to-use, flexible and inexpensive—prices range from just \$195 to \$825 in unit quantities.

The main members of the SBC-80 family are the 80/04, 80/05, 80/10A, 80/20 and 80/20-4 central-processor boards, with either an 8080A or 8085 microprocessor acting as the master CPU (Table 1). Most of the boards measure 6.75×12 in. and contain the CPU, clock, read/write memory, control ROM, I/O ports, serial communications interface and bus-control logic.

I/O interfacing is an area where design flexibility is essential to meet changing requirements efficiently. The programmable parallel and serial I/O structures of the boards make them versatile enough to do just that. What's more, upgrading system performance is easy thanks to the SBC-80 system bus, the Multibus, which permits modular performance expansion.

The Multibus provides a defined, standard interface between the SBC-80 single-board computers and expansion boards. As many as 16 SBC-80 family boards can simultaneously share the bus.

All in the SBC-80 family

As exemplified by the block diagram of the SBC-80/10A (Fig. 1), the SBC-80 microcomputer system has all that's needed for many applications. The SBC-80/10A is the oldest board in the family and has been widely imitated since it was one of the first "standardized" microcomputers commercially available.

The CPU section of the 80/10A board consists of

George Adams, Product Line Manager, Single-Chip Microcomputers, Intel Corp., 3065 Bowers Ave., Santa Clara, CA 95051.

Note: Multibus, RMX-80, ICE and Inteltec are registered trademarks of Intel Corp.

Reprinted from **ELECTRONIC DESIGN**/February 1, 1978

Copyright Hayden Publishing Co., Inc. 1978. All rights reserved.

the 8080A CPU, the 8224 clock generator and the 8238 system controller. Capable of fetching and executing any of the 8080A's 78 instructions, the CPU section can respond to interrupt requests originating on and off the board. (For more about the 8080A, see "Microprocessor Basics, Part 2," ED No. 10, May 10, 1976, p. 84).

The system-bus interface section includes an assortment of circuits to gate the interrupt and hold requests, the ready signals, and a system-reset signal. Other circuits drive the various control lines. Two 8216s help drive the bidirectional data bus, and six 8226s drive the external system-data and address buses as part of the SBC-80/10A's Multibus interface.

The RAM section of the 80/10A consists of 1024 bytes of static MOS memory. For program storage, up to 8192 bytes of ROM can be mounted on the board in 1024-byte increments by means of a 2708 or 8708 EPROM, an 8308 mask-programmed ROM, or in 2048 byte increments via the 2716 EPROM or 2316 ROM.

A serial interface on the board uses an 8251 programmable universal synchronous/asynchronous receiver/transmitter to provide a serial-data channel. The serial port operates at programmable rates up to 38,400 baud (synchronously) or 19,200 baud (asynchronously) with a choice of character length, number of stop bits, and even, odd or no parity. On-board interfaces provide direct EIA RS-232 or teletypewriter current-loop compatibility.

Two 8255 programmable peripheral interface circuits provide 48 I/O lines for transferring data to or from peripheral devices. Eight already-committed lines have bidirectional drivers and termination networks permanently installed, so that they can be inputs, outputs or bidirectional (jumper-selectable). The other 40 lines are uncommitted. On-board sockets permit drivers and termination networks to be installed, as needed. Since software configures the I/O lines, I/O can be customized for every application.

The 80/10A also responds to a single-level interrupt that can originate from one of many sources, the USART, programmable I/O and two user-designated interrupt-request lines. When an interrupt is recognized, a Restart-7 instruction is generated, and the processor accesses location 38_H to get the starting address of the service routine.

System expansion and support are possible with a

wide variety of alternate-source CPU, memory, and I/O boards (Tables 2 and 3). Up to 65,536 bytes of ROM, PROM or RAM can be accessed by one 80/10A. Expandable backplanes and card cages are also available to support multiboard systems.

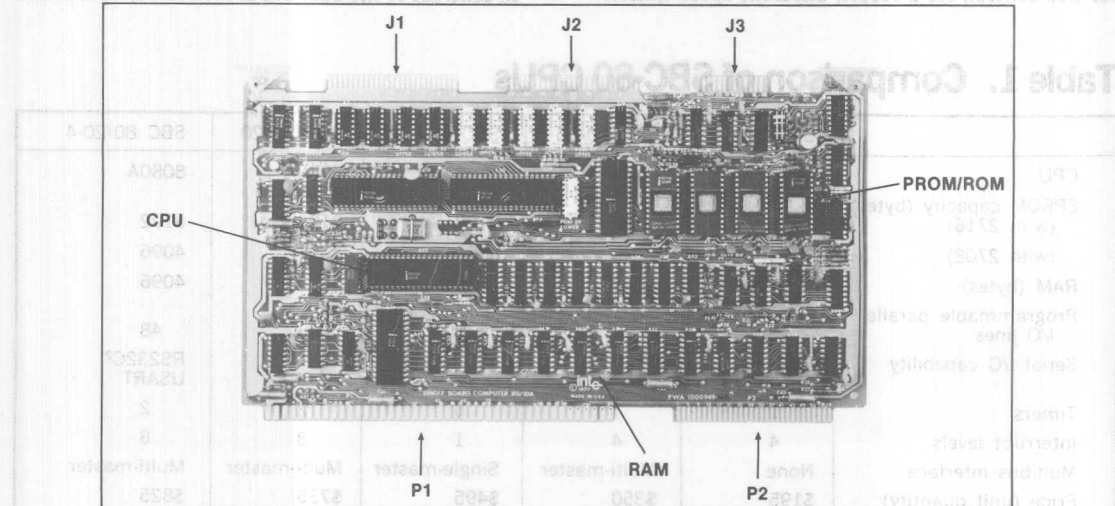
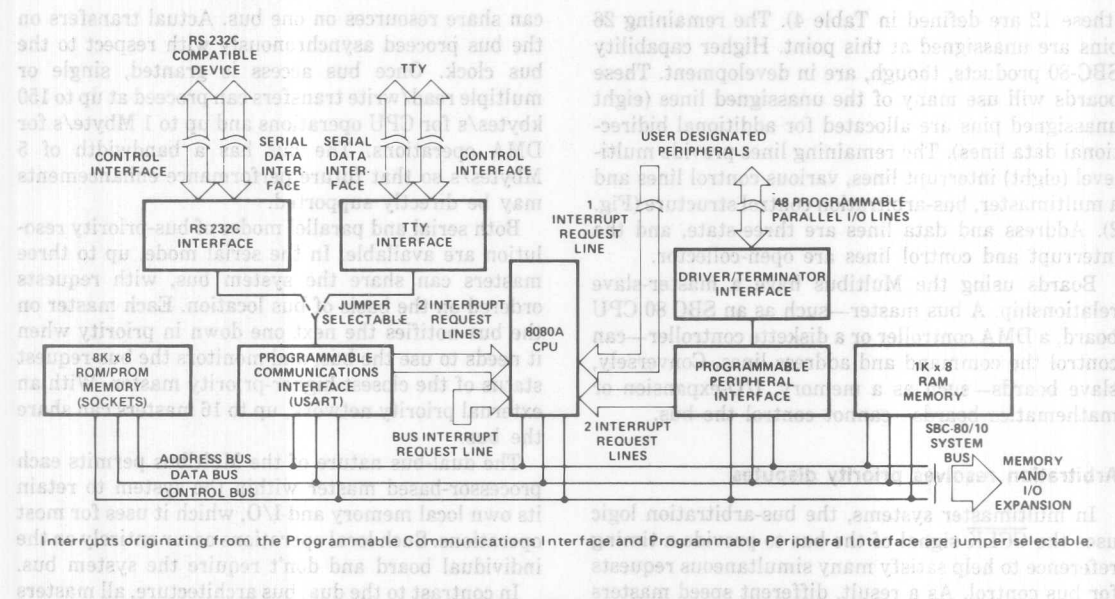
Interfacing starts with the bus

Although the SBC-80/10A is a complete microcomputer system, it can be expanded readily or it can serve as a primary master controller for other microcomputer cards. The 80/10A has five edge connectors, three on the top of the board and two on the backplane, or bottom, side. Two of the "top" connectors, J₁ and J₂, serve as parallel I/O ports, while J₃ is a serial I/O

port. All parallel I/O lines on the 50-contact J₁ and J₂ connector areas are paired with an independent signal/ground pin to permit alternate signal/ground wiring when using flat-cable interconnects. Serial port J₃ uses a 26-contact PC-edge connector to provide interfaces for both RS-232 and current-loop devices.

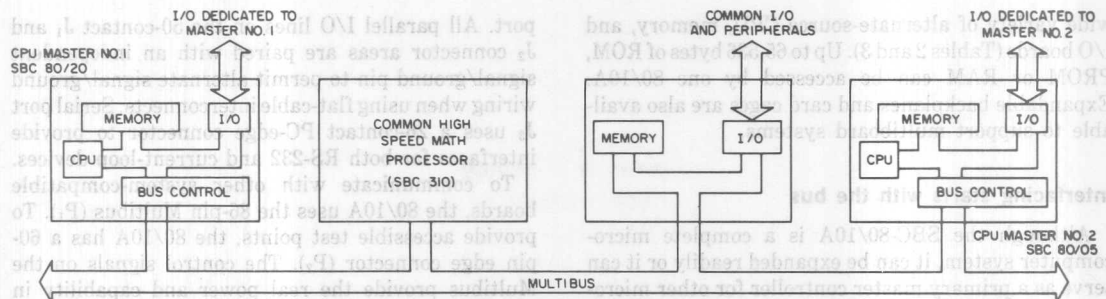
To communicate with other system-compatible boards, the 80/10A uses the 86-pin Multibus (P₁). To provide accessible test points, the 80/10A has a 60-pin edge connector (P₂). The control signals on the Multibus provide the real power and capability in control applications.

Of the 86 pins that make up the Multibus, 24 are assigned to power and ground, 16 to addressing, eight to bidirectional data, and 12 to signal and control



1. Based on an 8080A μ P, the 80/10A microcomputer has a straightforward design suitable for general-purpose

computing and control. The board has 48 programmable I/O lines and serial interfaces.



2. The Multibus interface for the SBC-80 CPU boards not only permits simultaneous multiprocessing, but also enables several processors to share the same bus and

peripheral devices. Arbitration logic on the CPU boards decides which board gets on the bus first if several units simultaneously access the bus.

(these 12 are defined in Table 4). The remaining 26 pins are unassigned at this point. Higher capability SBC-80 products, though, are in development. These boards will use many of the unassigned lines (eight unassigned pins are allocated for additional bidirectional data lines). The remaining lines provide multi-level (eight) interrupt lines, various control lines and a multimaster, bus-arbitration control structure (Fig. 2). Address and data lines are three-state, and the interrupt and control lines are open-collector.

Boards using the Multibus have a master-slave relationship: A bus master—such as an SBC 80 CPU board, a DMA controller or a diskette controller—can control the command and address lines. Conversely, slave boards—such as a memory, I/O-expansion or mathematics boards—cannot control the bus.

Arbitration resolves priority disputes

In multimaster systems, the bus-arbitration logic uses the $\overline{\text{CCLK}}$ signal of the bus to provide a timing reference to help satisfy many simultaneous requests for bus control. As a result, different speed masters

can share resources on one bus. Actual transfers on the bus proceed asynchronously with respect to the bus clock. Once bus access is granted, single or multiple read/write transfers can proceed at up to 150 kbytes/s for CPU operations and up to 1 Mbyte/s for DMA operations. The bus has a bandwidth of 5 Mbytes/s so that future performance enhancements may be directly supported.

Both serial and parallel modes of bus-priority resolution are available. In the serial mode, up to three masters can share the system bus, with requests ordered on the basis of bus location. Each master on the bus notifies the next one down in priority when it needs to use the bus, and monitors the bus-request status of the closest higher-priority master. With an external priority network, up to 16 masters can share the bus.

The dual-bus nature of the Multibus permits each processor-based master within the system to retain its own local memory and I/O, which it uses for most operations. Such local operations occur entirely on the individual board and don't require the system bus.

In contrast to the dual bus architecture, all masters

Table 1. Comparison of SBC-80 CPUs

	SBC 80/04	SBC 80/05	SBC 80/10A	SBC 80/20	SBC 80/20-4
CPU	8085	8085	8080A	8080A	8080A
EPROM capacity (bytes)					
(with 2716)	4096	4096	8192	8192	8192
(with 2708)	2048	2048	4096	4096	4096
RAM (bytes)	256	512	1024	2048	4096
Programmable parallel I/O lines	22	22	48	48	48
Serial I/O capability	RS232C SID/SOD ^{1, 2}	RS232C SID/SOD ^{1, 2}	RS232C/TTY USART	RS232C ² USART	RS232C ² USART
Timers	1	1	0	2	2
Interrupt levels	4	4	1	8	8
Multibus interface	None	Multi-master	Single-master	Multi-master	Multi-master
Price (unit quantity)	\$195	\$350	\$495	\$735	\$825

Notes: ¹Provided by 8085 CPU SID and SOD serial I/O lines. ²Optional SBC 530 TTY interface is available.

Table 2. Additional SBC support boards

Function	Model	Description	Price (unit qty)
RAM	SBC 016	16 kbyte dynamic RAM	\$ 825
	SBC 032	32 kbyte dynamic RAM	\$1360
	SBC 048	48 kbyte dynamic RAM	\$1860
	SBC 064	64 kbyte dynamic RAM	\$2200
	SBC 094*	4 kbyte CMOS static RAM with 96 hour battery backup.	\$ 795
EPROM	SBC 416	16 kbytes using 2708 type (1024 x 8) EPROM	\$ 295
Digital I/O	SBC 508*	32 input lines/32 output lines, all buffered/terminated	\$ 350
	SBC 517	48 programmable parallel lines with full buffering/termination options, full RS232C port, 1 ms real-time clock, and 8-line interrupt control	\$ 400
	SBC 519*	72 programmable parallel lines with full buffering/termination options, real-time clock (interval is jumper selectable to 0.5, 1, 2, or 4 ms), and 8-level programmable interrupt control.	\$ 395
Communications	SBC 534	Four programmable synchronous/asynchronous serial ports, each with: programmable baud rates, programmable data formats, programmable interrupt control, 16 RS232C buffered programmable parallel I/O lines configured as a Bell Model 801 automatic calling unit interface. Two programmable 16-bit interval timers (usable as real-time clocks), and software selectable loop-back of serial ports for diagnostic use.	\$ 650
	SBC 556*	48 optically isolated lines: 24 input 16 output, and 8 programmable (in/out), 8-level programmable interrupt control, and 1 ms real-time clock.	\$ 395
Analog I/O	SBC 711*	16/8 (single-ended/differential) 12-bit a/d channels; user expandable on-board to 32/16 channels	\$ 895
	SBC 724*	Four 12-bit d/a channels	\$ 750
	SBC 732*	Combination analog I/O; same a/d capability as SBC 711 plus 2 d/a channels	\$1125
Combination memory and I/O	SBC 104	8 kbytes capacity (sockets) using 2716 (2 k x 8) EPROM or 4 k using 2708, 4 kbytes dynamic RAM, 48 programmable parallel I/O lines, with full buffering/termination, as options. RS-232C port, a 1 ms real-time clock, and an eight-line interrupt control	\$ 715

*Requires +5 V only.

in multimaster/single-bus systems use the common bus for all instruction or data fetches or whenever data must be written to output devices or memory. Rapidly, then, the system bus becomes the bottleneck for over-all system throughput. Masters in SBC-80 systems only use the Multibus when data or instructions are resident in common, or global, memory or I/O. Since masters can request the Multibus simultaneously, on-board arbitration logic resolves any multiple contention.

Examine board performance

A look at the entire family of SBC-80 microcomputers reveals varied levels of performance. All five boards are inexpensive, but the most inexpensive is the 80/04, which costs \$99 in 100-unit quantities, and is intended for stand-alone applications. To get the cost down, the board was designed to use the 8085 CPU and the 8155 RAM, timer and I/O circuit.

The 80/04 contains an 8085 CPU, 256 bytes of RAM, space for up to 4 kbytes of EPROM (two 2716 EPROMs, or two 2708 EPROMs), 22 programmable parallel I/O lines with sockets for buffer and termination options, a 14-bit programmable timer/event counter, and provision for an RS-232-C serial port using the 8085 SID/SOD serial interface. The board can also house an on-board +5-V regulator, so an unregulated voltage can be connected.

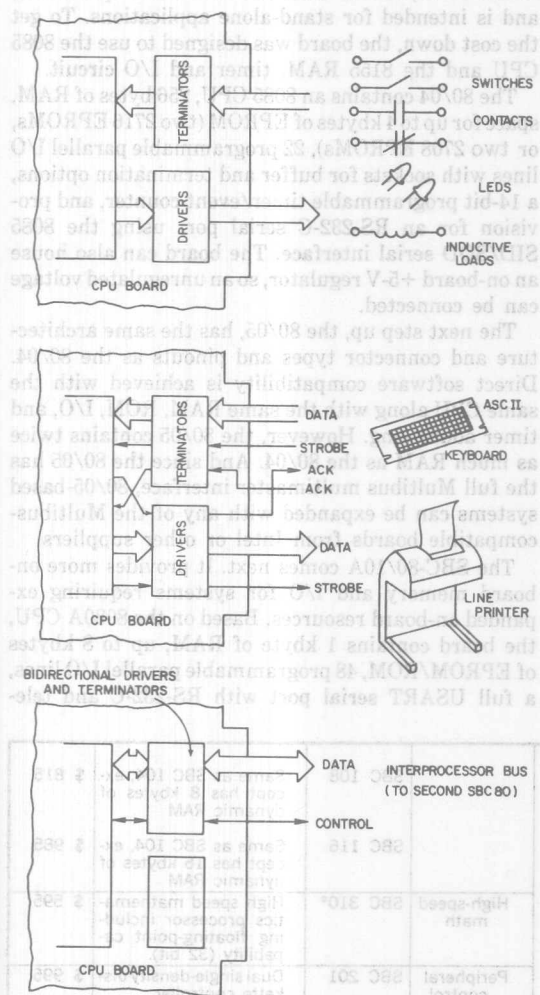
The next step up, the 80/05, has the same architecture and connector types and pinouts as the 80/04. Direct software compatibility is achieved with the same CPU along with the same RAM, ROM, I/O, and timer addressing. However, the 80/05 contains twice as much RAM as the 80/04. And since the 80/05 has the full Multibus multimaster interface, 80/05-based systems can be expanded with any of the Multibus-compatible boards from Intel or other suppliers.

The SBC-80/10A comes next. It provides more on-board memory and I/O for systems requiring expanded on-board resources. Based on the 8080A CPU, the board contains 1 kbyte of RAM, up to 8 kbytes of EPROM/ROM, 48 programmable parallel I/O lines, a full USART serial port with RS-232-C and tele-

	SBC 108	Same as SBC 104, except has 8 kbytes of dynamic RAM	\$ 815
	SBC 116	Same as SBC 104, except has 16 kbytes of dynamic RAM	\$ 985
High-speed math	SBC 310*	High speed mathematics processor including floating-point capability (32 bit).	\$ 595
Peripheral control	SBC 201	Dual single-density diskette controller	\$ 995
	SBC 202	Quad double-density diskette controller	\$1290
DMA control	SBC 501	DMA controller, up to 1 MHz transfer rates	\$ 450

typewriter interfaces, and a full Multibus interface (but only single-master capability; the board has no multimaster capability). Intended for single-CPU systems with only one other Multibus peripheral controller, the 80/10A can interface with such as the SBC-201 or SBC-202 single and double-density diskette controllers, or the SBC-501 DMA controller.

System designers requiring the same on-board I/O capability as the SBC-80/10A but with more RAM, more efficient real-time capability, and full multimaster Multibus control can go further up the ladder to the SBC-80/20 or SBC-80/20-4. These boards differ only in that the 80/20 contains 2 kbytes of RAM and the 80/20-4 contains 4 kbytes. Both boards can hold up to 8192 bytes of ROM or EPROM, handle up



3. Programmable I/O lines from the SBC-80 parallel interfaces can be set so that they are individually programmable as inputs or outputs (a), byte-programmable as inputs or outputs with handshaking (b), or bidirectional on a byte-programmable basis (c).

to eight levels of prioritized interrupt, and share the Multibus in the multimaster mode. Either board has two programmable interval timers/event counters. Auxiliary power buses and memory-protect control logic on the board permit battery backup of the RAM.

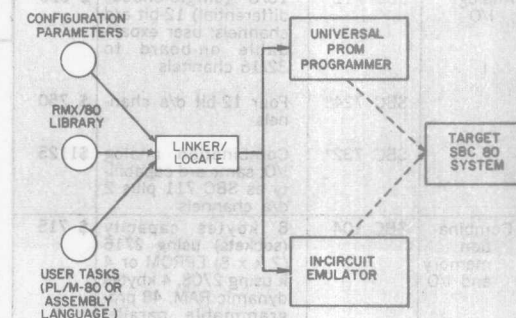
Take advantage of interrupts and timers

Real-time applications frequently require that high-priority programs operate on the basis of external events, time-of-day, or elapsed time without impacting current background processing. These multi-programming requirements are supported in the 80/20 and 80/20-4 by an eight-level programmable interrupt controller (PIC) and two programmable interval timer/event counters. The priority level of any event generating an interrupt request is assigned through jumper selection and the priority algorithm chosen by system software.

Any combination of interrupt levels may be masked by storing a single byte in the interrupt-mask register contained by the PIC, whose four software-selectable priority algorithms are described in Table 5. The PIC generates a unique memory address for each interrupt level. These addresses are equally spaced at intervals of 4 or 8 bytes (software-selectable). The resulting 32 or 64-byte block may begin at any 32 or 64-byte boundary in the 65,536-byte memory space. A single 8080A jump instruction at each of these addresses then provides linkage to locate each interrupt service routine independently anywhere in memory.

The two programmable timers may be used to generate real-time clocks by requesting periodic interrupts through the PIC, so that the CPU is free to handle numerous other system-timing and control functions. The outputs and gate/trigger inputs of the timer/counters can be routed via jumpers to the PIC, the I/O driver/terminators, or the programmable parallel I/O.

Seven software-selectable timing/counting functions are available. Either timer may be set to act as a rate generator (divide-by-N counter), a square-wave



4. By using the RMX-80 executive and the library of often-used subroutines, program development can be simplified since the subroutines are modular and can be linked together, then checked out in a system prototype.

Table 3. Non-Intel SBC-compatible boards

Manufacturer	BPM	BCLK	Multibus CPU boards	RAM boards	Core memory boards	PROM/ROM boards	Analog boards	AC power control boards	IEEE-488 bus	Tape interface	Floppy-disc controller	Communications interface	Video boards	Math processors	Breadboards	Circle no.
ADAC Corp., 118 Cummings Park, Woburn, MA 01801 (617) 935-6668																451
Ampex, Memory Products Div., 200 N. Nash St., El Segundo, CA 90245 (213) 640-0150																452
Analog Devices, Route 1 Industrial Park, P.O. Box 280, Norwood, MA 02062 (617) 329-4700																453
Augat Inc., 33 Perry Ave., P.O. Box 779, Attleboro, MA 02703 (617) 222-2202																454
Burr-Brown, International Airport Industrial Park, P.O. Box 11400, Tucson, AZ 85734 (602) 294-1431																455
Cybernetic Microsystems, 2460 Embarcadero Way, Palo Alto, CA 94303 (415) 321-0410																456
Data Translation Inc., 23 Strathmore Road, Natick, MA 01760 (617) 655-5300																457
Datacube Corp., 25 Industrial Park, Chelmsford, MA 01824 (617) 256-2555																458
Datel Systems Inc., 1020 Turnpike St., Building S., Canton, MA 02021 (617) 828-8000																459
Digidata Corp., 8580 Dorsey Run Road, Jessup, MD 20794 (301) 498-0200																460
EDAC Corp., 1417 San Antonio Ave., Alameda, CA 94501 (415) 521-6600																461
Electronic Engineering & Prod. Services, TE. #2, Louisville, TN 37777 (615) 984-9640																462
Electronic Solutions, 7969 Engineer Rd., San Diego, CA 92111 (714) 292-0242																463
Garry Mfg. Co., 1010 Jersey Ave., New Brunswick, NJ 08902 (201) 545-2424																464
Hal Communications Corp., Box 365B, 807 E. Green St., Urbana, IL 61801 (217) 367-7373																465
Iasis, 815 W. Maude Ave., Sunnyvale, CA 94086 (408) 732-5700																466
ICOM, 6741 Variel Ave., Canoga Park, CA 91303 (213) 348-1391																467
Megalogic Corp., 9650 National Road, Brookville, OH 45309 (513) 833-5222																468
Micro Memories Inc., 9438 Irondale Ave., Chatsworth, CA 91311 (213) 998-0700																469
Microtec, P.O. Box 60337, Sunnyvale, CA 94088 (408) 733-2919																470
Monolithic Systems Inc., 14 Inverness Drive, East, Englewood, CA 80110 (303) 770-7400																471
National Semiconductor, 2900 Semiconductor Drive, Santa Clara, CA 95051 (408) 737-5000																472
North Star Computers Inc., 2465 Fourth St., Berkeley, CA 94710 (415) 549-0858																473
The Thomas Engineering Co., 1201 Park Ave., Emeryville, CA 94608 (415) 547-5860																474
Vector Electronic Products, 12460 Gladstone Ave., Sylmar, CA 91342 (213) 365-9661																475
Zia Tech., 10762 La Roda Drive, Cupertino, CA 95015 (408) 996-7082																476

generator, a programmable retriggerable one-shot, or software or hardware-triggered strobe. One of the timers can be jumper-selected as an event counter, and either can generate an interrupt after a specified interval or after a specified number of events.

The programmability of each on-board timer allows timing intervals from approximately 2 μ s to over 60 ms. But the two timers may be cascaded to provide intervals greater than 1.1 hour, in 1.86 μ s increments. In the event counter mode, external event rates up to 1.1 MHz may be counted.

Flexible I/O, a must for any system

All SBC-80 microcomputers provide 22 or 48 programmable parallel I/O lines that, grouped as 8-bit ports, are fully programmable to allow enough flexibility to handle any changes in system interfacing. Programmability is permitted through data direction, control mode, interrupt handling, and buffer/termination. The I/O configuration for a specific application is selected through software initialization of the parallel I/O control logic, jumper selection of control/interrupt line routing, and the particular buffer and termination devices chosen.

Fig. 3 illustrates the basic modes of operation that may be selected by software to meet application requirements. Mode 0 is used for slow-to-medium-speed interfacing where immediate handshake response or interrupt generation is not needed. This mode is extremely useful for interfacing to inputs such as switches or outputs such as LED indicators or numeric displays.

Mode 1 provides handshaking lines required for many medium to high-speed peripherals. A typical output function could be a line printer; an input device could be an encoded keyboard or paper tape reader.

In addition, the 80/10A and 80/20 have Mode 2, a bidirectional data/control structure. This interface may provide, for example, a communication link between parallel processors.

The SBC-80 I/O structure also permits multiple options for output buffering and input termination. TTL drivers with 16 to 48 mA of drive can be used, and input lines may be terminated to minimize the impact of noise and cable disconnects. Any of the TTL drivers (four outputs) or input terminators (for inputs) listed in Table 6 may be inserted into sockets to provide proper buffering or termination.

Like the design flexibility of the SBC-80 parallel I/O structure, the serial I/O structure allows interface characteristics to be revised rapidly through software, jumper, and buffer changes. Besides the SBC-80/10A, the 80/20 and 80/20-4 contain the USART serial channel. These boards provide RS-232 interfaces, but the SBC-80/10A also has a teletypewriter current-loop interface. Synchronous/asynchronous mode, data format, control-character format, and parity are all under program control. So is baud rate on the 80/20 and 80/20-4. Baud rate is jumper-selectable on the

Table 4. Multibus control signals

AACK	Advance-acknowledge signal, used in 8080A-based systems. It is sent to the SBC-80 board by a memory bank in response to a memory-read command, allowing the memory to complete the access without requiring the CPU to wait.
BCLK	Bus clock, used to synchronize bus-control circuits on all master boards. It has a period of 101.725 ns (9.8304 MHz) and a 30 to 70% duty cycle. The signal may be slowed, stopped or single-stepped.
BPRN	Bus-priority-input signal, used to indicate to the master that a higher-priority master board wants to use the system bus. When brought high, the signal suspends processing activity and places line drivers of the master in a standby mode.
BUSY	Bus-busy signal, a bidirectional control line that allows control and monitoring of the Multibus in multimaster systems. As an output from a bus master, BUSY indicates the bus is being used by the board. It prevents all other master boards from gaining control of the bus. Each master monitors BUSY as an input to determine current Multibus usage status.
CCLK	Constant clock, used to provide a 9.8304-MHz clock signal for optional memory and I/O expansion boards. CCLK coincides with BCLK and has a period of 101.725 ns and a 30 to 70% duty cycle.
INIT	Initialize signal, used to reset the entire system to a known internal state.
INTR1	Interrupt input, used to interrupt the processor via an externally generated interrupt request.
IORC	I/O-read command, a signal generated by the master to indicate that the address of an input port has been placed on the system-address bus and that the data at that input port are to be read and placed on the system-data bus.
IOWC	I/O-write command, a signal generated by the master to indicate that the address of an output port has been placed on the system-address bus and that the contents of the system-data bus are to be output to the addressed port.
MRDC	Memory-read command, a signal generated by the master that indicates that the address of a memory location has been placed on the system-address bus. It specifies that the contents of the addressed location are to be read and placed on the system-data bus.
MWTC	Memory-write command, a signal generated by the master to indicate that the address of a memory location has been placed on the system-address bus. It causes information on the data bus to be written into the addressed memory location.
XACK	Transfer-acknowledge signal, an input signal to the master board from an external memory location or I/O port to indicate that a specified read or write operation has been completed.

80/10A CPU board.

The synchronous and asynchronous nature of the serial interface makes it compatible with virtually every standard serial data-transmission technique used today (including IBM's Bi-Sync). This allows multiple SBC-80 boards to be interconnected as a distributed-processing network. The resulting task segregation or redundancy (or both) significantly improves both system performance and reliability.

Two jumper-selectable interrupt requests may be generated automatically by the serial interface. One occurs when a newly received character is ready to be loaded into the CPU (receive-channel buffer is full). The other occurs when new data are ready to be transmitted to the remote device (transmit-data buffer is empty).

Both the SBC-80/04 and 80/05 provide serial I/O capability through the serial input data (SID) and serial output data (SOD) functions of the 8085 CPU. These functions are controlled by software executing the 8085 read-interrupt mask (RIM) and set-interrupt mask (SIM) instructions.

For systems requiring many serial channels, the SBC-534 communications-expansion board provides four USART channels with RS-232-C and optically isolated current-loop interfaces, programmable interrupt, timing, baud-rate control, and a Bell 801 Auto-Call unit interface.

Expand the system via the Multibus

The SBC-80 family is gaining not only in popularity but in support for its Multibus as more and more companies offer SBC-compatible boards. Intel now provides high-speed mathematics, RAM, EPROM, mass storage, digital I/O, combination memory and I/O, serial communications, and analog-I/O expansion boards.

For applications requiring fast, high-precision number crunching, the SBC-310 math unit acts as an intelligent slave to perform floating-point and fixed-point mathematics. A processor uses the 310 by passing parameters to it along with a command byte to select the desired operation from the SBC-310's 14 instructions. The repertoire includes 32-bit floating-point (single-precision) addition, subtraction, multiplication, division, squaring, square root, comparisons, and tests; 16-bit fixed-point multiply, subtract, extended divide, and extended compare; and conversion from fixed to floating point or vice versa.

A completed operation may be signaled either by the math unit via an interrupt or by the host processor's polling the "operation complete" flag in the unit's status register. The result may be retrieved at this point or left in the 310's accumulator for further use.

In addition, the 310 provides control circuitry so that it may be treated as a "shared resource" among several CPU boards.

Two diskette options are available for mass storage.

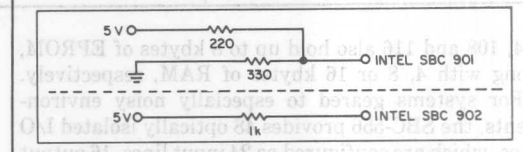
Table 5. Programmable interrupt modes, SBC-80/20-4

Mode	Operation
Fully nested	Interrupt request line priorities fixed at 0 as highest, 7 as lowest.
Autorotating	Equal priority. Each level, after receiving service, becomes the lowest priority level until next interrupt occurs.
Specific priority	System software assigns lowest priority level. Priority of all other levels based in sequence on this assignment.
Polled	System software examines priority-encoded system interrupt status via interrupt status register.

Table 6. Line drivers and terminators

Line drivers		
Driver	Characteristic	Sink current (mA)
7438	I, OC	48
7437	I	48
7432	NI	16
7426	I, OC	16
7409	NI, OC	16
7408	NI	16
7403	I, OC	16
7400	I	16

Note: I = inverting; NI = noninverting; OC = open collector



The SBC-201 diskette controller provides full control for one or two single-density diskette drives and acts as a programmable slave to masters on the Multibus. All diskette information is stored in the IBM soft-sectored format. For systems requiring greater storage capacity, the SBC-202 provides full control for up to four double-density diskette drives. Thus, 2 Mbytes of mass storage may be added to SBC-80-based systems for each SBC-202 plugged into the bus.

Digital I/O may be expanded using any of three Intel boards. The SBC-519 provides 72 programmable parallel I/O lines as well as interrupt handling and a real-time clock.

The 519's clock can interrupt the appropriate CPU periodically so that the CPU can monitor system-I/O status. High-speed I/O events can gain the CPU's attention via interrupts. The SBC-517 combination I/O board and the SBC-104, 108 and 116 combination memory and I/O boards offer 48 programmable parallel lines, a full RS-232 USART serial channel, interrupt handling and a 16-ms real-time clock. The

Table 7. RMX-80 routine library

RMX/80 module	Function
Nucleus (executive)	Provides basic capabilities (concurrency, priority, and synchronization/communication) found in all real-time systems.
Terminal handler	Provides real-time asynchronous I/O between an operator's terminal and tasks running under the RMX/80 executive, includes a line-edit feature similar to that of ISIS-II (supervisory system on the Intellec development system) and type-ahead facility.
Diskette file systems	Diskette driver and file management capabilities, allows user to load tasks into the system and to create, access, and delete files in a real-time environment without disrupting normal processing. File formats compatible with ISIS-II for both single and double-density systems.
Free space manager	Maintains a pool of free RAM and allocates memory out of the pool upon request from a task; reclaims memory areas when no longer needed.
Debugger	Specifically designed for debugging software running under the RMX/80 executive; used by linking it to an application program or task. Thus, it can be run directly from the single-board computer's memory.
Math handler	Provides full control and communication for SBC 310 math board for high-speed fixed and floating-point math functions.
Analog interface handler	Provides real-time control for SBC 711, 724, and 732 analog I/O expansion boards.

104, 108 and 116 also hold up to 8 kbytes of EPROM, along with 4, 8 or 16 kbytes of RAM, respectively.

For systems geared to especially noisy environments, the SBC-556 provides 48 optically isolated I/O lines, which are configured as 24 input lines, 16 output lines, and eight programmable-I/O lines. The user fixes the optical-isolation characteristics according to his exact system requirements by installing the optoisolators and current-limiting resistors of his choice into the board sockets. Input voltages up to 48 V, output lines up to 30 V and currents up to 60 mA may be interfaced.

Of course, many more RAM, ROM, communications and interface options are available. But for systems to come together quickly during development, there must be some standardized operating software to provide some of the most fundamental system routines.

System software: the glue that binds

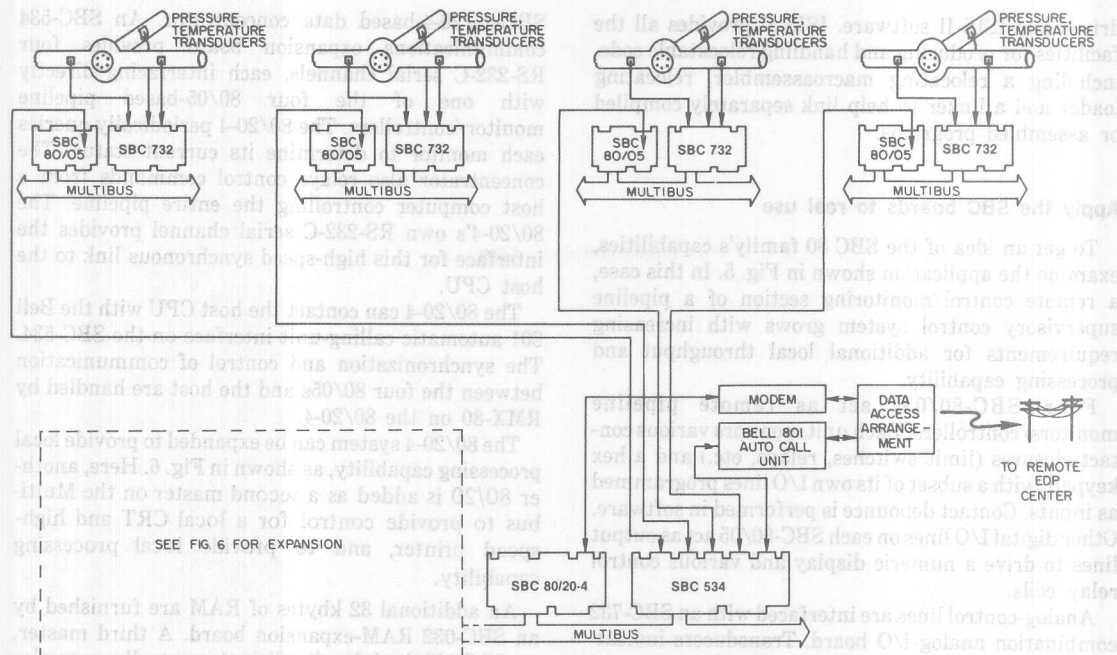
Where the Multibus provides a standard hardware structure, RMX-80, a real-time multitasking executive supplies a modular software framework. With RMX-80, routines don't have to be developed for task synchronization, priority resolution and peripheral control (printers, terminals, diskettes, etc.). Versions

are available for the SBC-80/20, 80/20-4 and 80/10A CPU boards.

Critical projects can be completed rapidly because RMX-80 provides major portions of the software requirements for many real-time systems. For example, the diskette file-extension software of the RMX-80 program may be linked into the system software. Thus, users can immediately have a diskette file structure with facilities to open and close files, create and delete files, read or write files sequentially or randomly (read function may be used for initial program load, if desired), or allocate file storage dynamically on single or double-density diskettes.

The compactness of RMX-80—the entire executive resides in 2 kbytes of ROM—reduces memory requirements and eliminates the need for bootstrap-program loading. All RMX-80 operations are based on individual tasks. A task is a program with unique data and stack that operates asynchronously with other such programs in the system.

Basically, the RMX-80 is a library of "standard" routines (Table 7), such as an analog-interface handler and a terminal handler. Fig. 4 illustrates how to develop software by selecting appropriate RMX-80 modules, then locating and linking them with particular software tasks on an Intellec microcomputer development system. In addition, a debugger module



5. This possible SBC-80 system configuration uses four SBC-80/05s to monitor and control pipeline parameters

in the RMX-80 speeds real-time system development.

The executive accesses system resources according to task priority, intertask communication, interrupt-driven control for standard devices, real-time clock control, interrupt handling, and other optional functions. In all, there are 255 separate task-priority levels, and since multiple tasks may share the same level, the actual number of tasks is limited only by memory size.

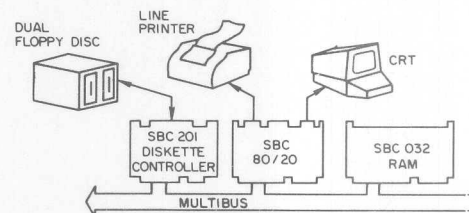
Develop programs with the Intellec

The Intellec and its ICE-80 and ICE-85 in-circuit emulators help minimize the time required to develop software and hardware. Standard Intellec stand-alone software includes resident macroassemblers for the 8080A and 8085 CPUs, a text editor, and a system monitor/debugger. As a result, programs can be assembled, loaded, edited, executed, and debugged.

ICE diagnostics can significantly reduce program development and debug time. Break points may be set on user-specified memory-read or write operations, I/O read or write operations, or user-defined extension parameters. Programs can be single-stepped to check operating conditions and performance.

PL/M-80 is the high-level systems-programming language. The optional Intellec-resident PL/M compiler provides the ability to program in this natural, algorithmic language, so there is no need to manage register usage or to allocate memory. PL/M programs

and feed data back to a master controller, an SBC-80/20-4. The master controller sends data back to a host system.



6. Expanding the pipeline monitor/controller system is as simple as plugging more cards into the Multibus and altering the software. By adding another SBC-80/20 to the master controller, some local processing can be done and a local CRT and high-speed printer can be added as well as RAM and diskette-memory space.

may be linked to assembly-language programs to hasten product development further.

A relocatable macroassembler residing on the Intellec translates symbolic assembly language into 8080 or 8085 machine code and permits the use of relocatable and linkable object code. With full macro capability, similar sections of code needn't be written over and over.

Intellec options include a diskette operating system, ISIS-II, with diskette controller, single or dual diskette

or assembled programs.

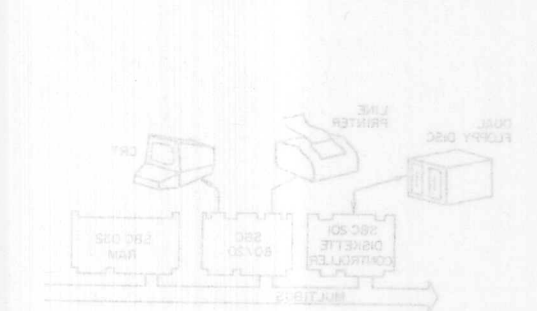
Apply the SBC boards to real use

To get an idea of the SBC 80 family's capabilities, examine the application shown in Fig. 5. In this case, a remote control/monitoring section of a pipeline supervisory control system grows with increasing requirements for additional local throughput and processing capability.

Four SBC-80/05s act as remote pipeline monitors/controllers. Each unit monitors various contact closures (limit switches, relays, etc.) and a hex keypad, with a subset of its own I/O lines programmed as inputs. Contact debounce is performed in software. Other digital I/O lines on each SBC-80/05 act as output lines to drive a numeric display and various control relay coils.

Analog-control lines are interfaced with an SBC-732 combination analog-I/O board. Transducers indicating temperature and pressure drive analog inputs, and analog outputs drive valves. Flow rate is determined in software by manipulating differential pressure data available from pressure transducers.

The four 80/05s are linked serially to a remote



Expanding the pipeline monitor/controller system is as simple as plugging more cards into the Multibus and altering the software. By adding another SBC-80/05 to the master controller, some local processing can be done and a local CRT and high-speed printer can be added as well as RAM and diskette-memory space.

may be linked to assembly-language programs to hasten product development further. A relocatable macroassembler residing on the Intel 8085 translates symbols, assembly language into 8085 or 8086 machine code and permits the use of relocatable and linkable object code. With full macro capability, similar sections of code needn't be written over and over.

Intellex options include a diskette operating system, iSIS-II, with diskette controller, single or dual diskette

monitor/controllers. The 80/20-4 periodically queries each monitor to determine its current status. The concentrator also relays control commands from a host computer controlling the entire pipeline. The 80/20-4's own RS-232-C serial channel provides the interface for this high-speed synchronous link to the host CPU.

The 80/20-4 can contact the host CPU with the Bell 801 automatic calling-unit interface on the SBC-534. The synchronization and control of communication between the four 80/05s and the host are handled by RMX-80 on the 80/20-4.

The 80/20-4 system can be expanded to provide local processing capability, as shown in Fig. 6. Here, another 80/20 is added as a second master on the Multibus to provide control for a local CRT and high-speed printer, and to provide local processing capability.

An additional 32 kbytes of RAM are furnished by an SBC-032 RAM-expansion board. A third master, an SBC-202 dual-density diskette controller, can also be added to the Multibus, along with two double-density diskette drives. Communication between the two 80/20s is handled via user-written intermaster message tasks.■

The executive manages system resources according to task priority, interrupt communication, interrupt-driven control for standard devices, real-time clock control, interrupt handling, and other optional functions. In all, there are 255 separate task-priority levels, and since multiple tasks may share the same level, the actual number of tasks is limited only by memory size.

Develop programs with the Intellex

The Intellex and its ICB-80 and ICB-85 in-circuit emulators help minimize the time required to develop software and hardware. Standard Intellex stand-alone software includes resident macroassemblers for the 8080A and 8085 CPUs, a text editor, and a system monitor/debugger. As a result, programs can be assembled, loaded, executed, and debugged. ICE diagnostics can significantly reduce program development and debug time. Break points may be set on user-specified memory-read or write operations, I/O read or write operations, or user-defined extension parameters. Programs can be single-stepped to check operating conditions and performance.

PLM-80 is the high-level systems-programming language. The optional Intellex-resident PLM compiler provides the ability to program in this natural, algorithmic language, so there is no need to manage register usage or to allocate memory. PLM programs

April, 1978

Design decision factors involved in developing multiple processor microcomputer systems include means of minimizing contention for system bus utilization. System applications detail the appropriate hardware and software considerations as related to single board computers in a multibus structure.

George Adams and Thomas Rolander
Intel Corporation, Santa Clara, California

Design Motivations for Multiple Processor Microcomputer Systems

George Adams and Thomas Rolander
Microcomputer Applications

DESIGN MOTIVATIONS FOR MULTIPLE PROCESSOR MICROCOMPUTER SYSTEMS

Design decision factors involved in developing multiple processor microcomputer systems include means of minimizing contention for system bus utilization. System applications detail the appropriate hardware and software considerations as related to single-board computers in a multimaster bus structure

George Adams and Thomas Rolander

Intel Corporation, Santa Clara, California

Large-scale integrated circuit technology has reduced the cost of central processors to such a low level that the previously avoided concept of applying multiple processors to meet system performance requirements has now become an attractive and viable alternative. Several key benefits accrue from such an approach. In addition to enhanced system performance (throughput), improved system reliability, and improved system realtime response, modular system expansion capabilities may be realized. Although designing such systems "from scratch" with microprocessor component families can be a complex system design task with many subtle pitfalls which can inhibit efficient system operation, the advent of second generation single-board computers, such as the Intel® SBC 80/05 and 80/20, has allowed multiple processor microcomputer systems to become off-the-shelf products.

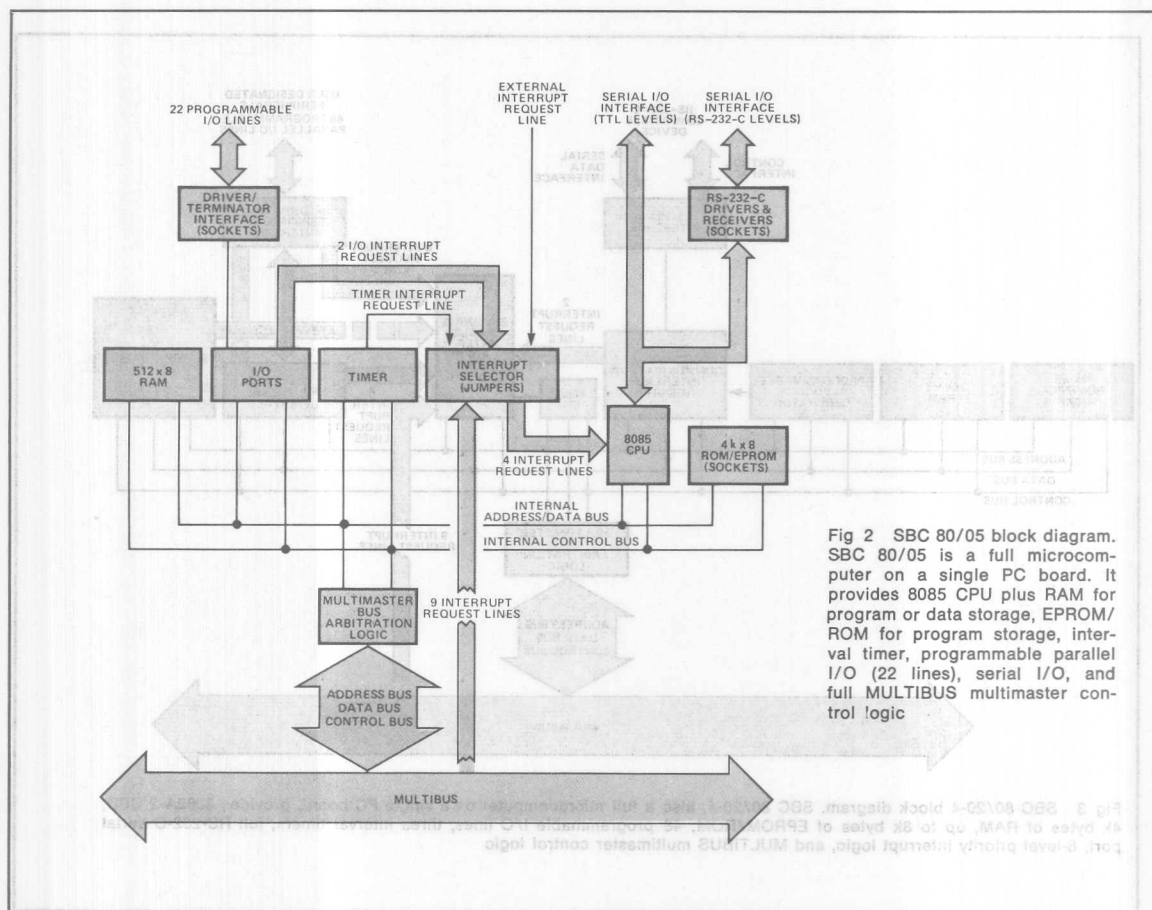
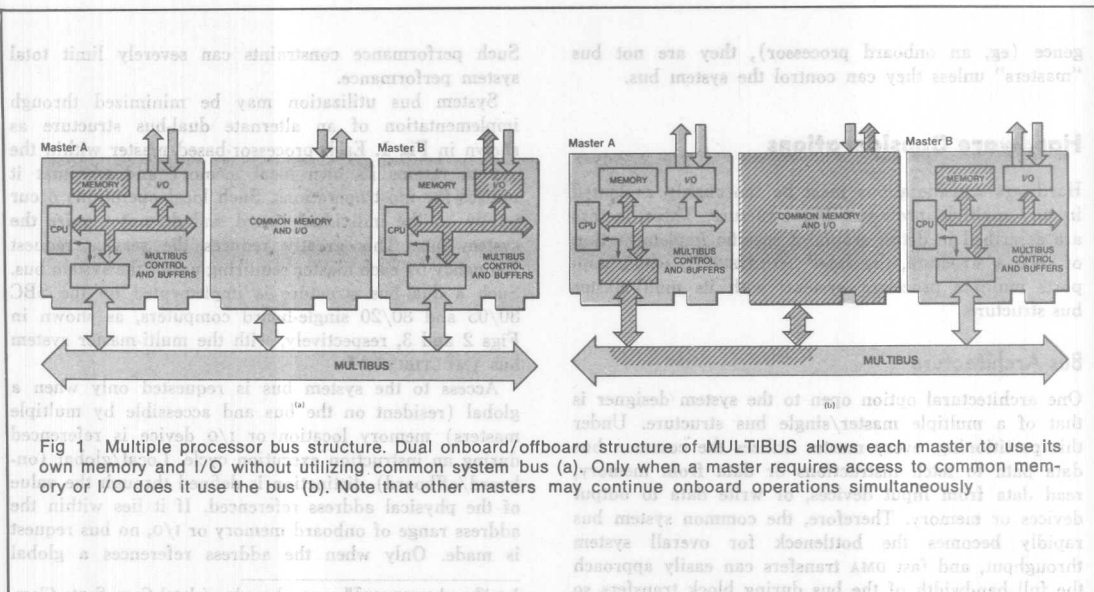
Motivation and Design Concepts

Discussion of the benefits of multiple processor structures in system applications will provide an understanding of the motivation for this implementation approach in system design. A primary objective addressed through

multiple processor approaches is enhanced system performance and throughput. Enhanced performance is achieved through partitioning of overall system functions into tasks that each of several processors can handle individually.

In general, as the number of individual tasks any given processor must handle is reduced, that processor's response time to new requests for service will be reduced. A well planned multiple processor bus structure will allow new processors to be added to the system in modular fashion. When new system functions (ie, more peripherals) are added, more processing power can be applied to handle them without impacting existing processor (master) task partitioning.

As used here, a "master" is any element existing on the system bus that may take control of the bus (ie, assert address and control lines). Typical examples include processors and direct memory access (DMA) controllers that address memory and input/output (I/O) locations resident on the bus. "Slave" elements include passive functions on the bus, such as memory or non-DMA I/O interfaces. Note that although slaves may possess intelli-



Hardware considerations must be thoroughly evaluated in any multiple processor bus structure. These factors are described in detail around a specific implementation of such a structure, the Intel® MULTIBUS™, which supports multiple processor systems with its multi-master bus structure.

Bus Architecture

One architectural option open to the system designer is that of a multiple master/single bus structure. Under this partitioning, every master utilizes the common bus data path to fetch instructions or data from memory, read data from input devices, or write data to output devices or memory. Therefore, the common system bus rapidly becomes the bottleneck for overall system throughput, and fast DMA transfers can easily approach the full bandwidth of the bus during block transfers so that all other masters must idle for extended periods.

system retains its own local memory and I/O that it utilizes for most operations. Such local operations occur totally on the individual board and do not require the system bus. This greatly reduces the service request frequency by each master requiring use of the system bus. Such a dual-bus structure is implemented on the SBC 80/05 and 80/20 single-board computers, as shown in Figs 2 and 3, respectively, with the multi-master system bus (MULTIBUS).^{1,2}

Access to the system bus is requested only when a global (resident on the bus and accessible by multiple masters) memory location or I/O device is referenced during an instruction execution cycle. Local/global (on-board/off-board) distinction is defined through the value of the physical address referenced. If it lies within the address range of onboard memory or I/O, no bus request is made. Only when the address references a global

Intel® and MULTIBUS™ are trademarks of Intel Corp, Santa Clara, Calif.

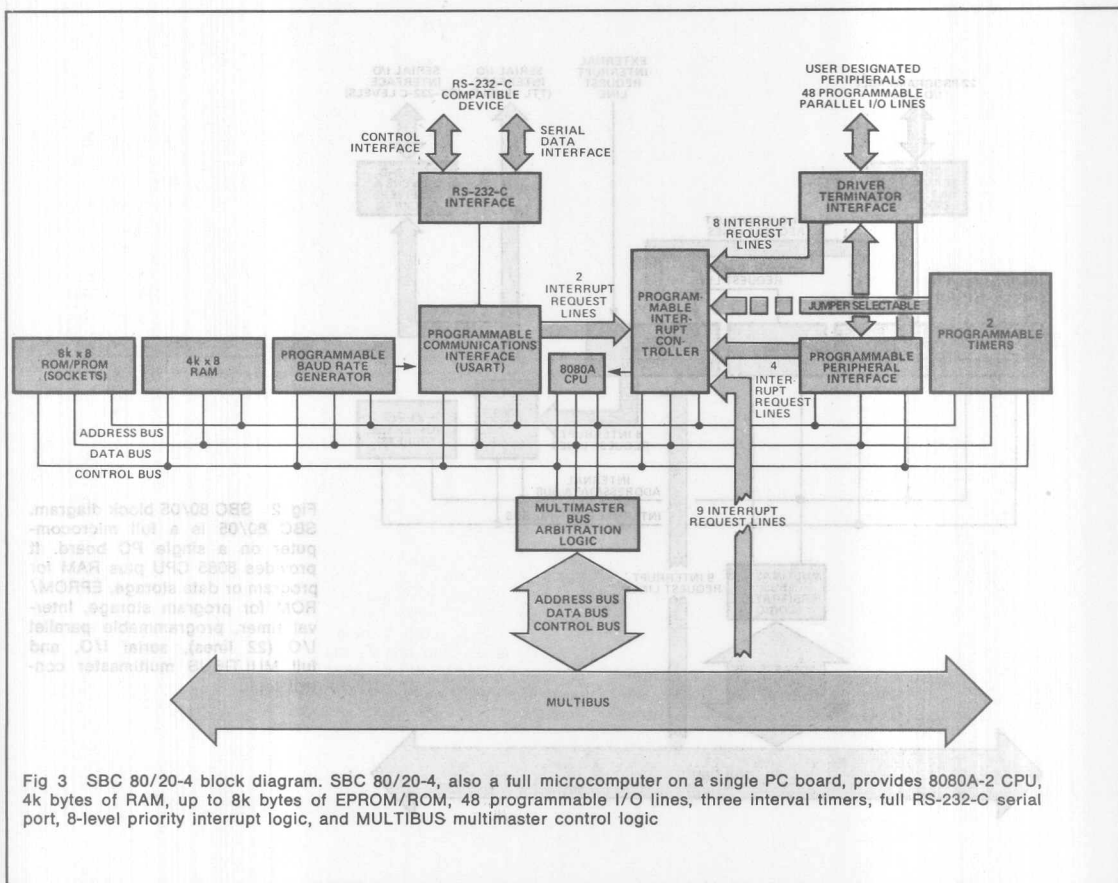


Fig 3 SBC 80/20-4 block diagram. SBC 80/20-4, also a full microcomputer on a single PC board, provides 8080A-2 CPU, 4k bytes of RAM, up to 8k bytes of EPROM/ROM, 48 programmable I/O lines, three interval timers, full RS-232-C serial port, 8-level priority interrupt logic, and MULTIBUS multimaster control logic

memory or I/O location, is a system bus request initiated. If no other master is currently utilizing the bus, this "new" master will be granted access immediately. However, this new master must wait if another master is currently utilizing the system bus. It continues to monitor the status of the system bus to determine when its current cycle may be completed. Thus, the MULTIBUS must provide a method for masters to determine whether or not another master is currently utilizing it.

Other masters may also simultaneously request the system bus. Arbitration must then be performed to resolve this multiple contention for the system bus. The MULTIBUS structure provides this arbitration in one of two techniques: serial (daisy chain) or parallel (encoded). The structure consists of four control lines that are synchronized by the common bus clock. These four control lines and the bus clock are active low. This is represented by the slash (/) character after each signal mnemonic. Control lines are as follows:

Bus Clock (BCLK/)—The negative edge of BCLK/ is used to synchronize bus arbitration. BCLK/ may be asynchronous to all CPU clocks, and it has a 100-ns minimum period. BCLK/ may be slowed, stopped, or single-stepped for debugging.

Bus Priority In Signal (BPRN/)—Indicates to a particular master that no higher priority master is requesting use of the system bus.

Bus Priority Out Signal (BPRO/)—Used with serial bus priority resolution scheme. BPRO/ is passed to BPRN/ input of master with next lower bus priority.

Bus Busy Signal (BUSY/)—Driven by bus master currently in control of MULTIBUS to indicate that bus is currently in use. BUSY/ prevents all other masters from gaining control of bus.

Bus Request Signal (BREQ/)—Used with parallel bus priority network to indicate that a particular master requires use of the bus for one or more data transfers.

Serial (Daisy-Chain) Bus Arbitration

In a serially arbitrated MULTIBUS system (Fig 4) requests for system bus utilization are ordered by priority on the basis of bus location. Each master on the bus notifies the next lower priority master when it needs to use the bus for a data transfer, and it monitors the bus request status of the next higher priority master. Thus the masters pass bus requests along from one to the next in a daisy-chain fashion.

The highest priority master (Master 1) in the system will always receive access to the system bus when it requires it. There is no higher priority master to inhibit its bus requests, and its bus priority input line (BPRN/) is thus permanently enabled.

Masters operate asynchronously on the MULTIBUS. A master may thus be in the middle of a bus operation when a higher priority master requests the bus. Obviously, interruption of such an in-process cycle must not be allowed. The mechanism for avoiding such erroneous operation is the BUSY/ line. Upon being notified that access to the bus is possible, the master examines BUSY/. If this control line is inactive, the master will assert it, and complete its bus operation. If BUSY/ is already active, another master is currently using the bus. In this case, the master will examine BUSY/ upon every falling edge of BCLK/, typically once every 100 ns, until it becomes inactive. When BUSY/ returns to its inactive state, the master will assert it, then complete its operation. The BUSY/ line then inhibits higher priority masters from destroying a bus transfer cycle that may be already in progress.

The BUSY/ line is also controlled by a bus lock function on the SBC 80/05 and 80/20. This function allows a master, which currently has control of the bus, to retain control by independently asserting the BUSY/ line until it issues an unlock command that releases BUSY/. This permits a bus master to obtain exclusive control of the system bus for critical system functions,

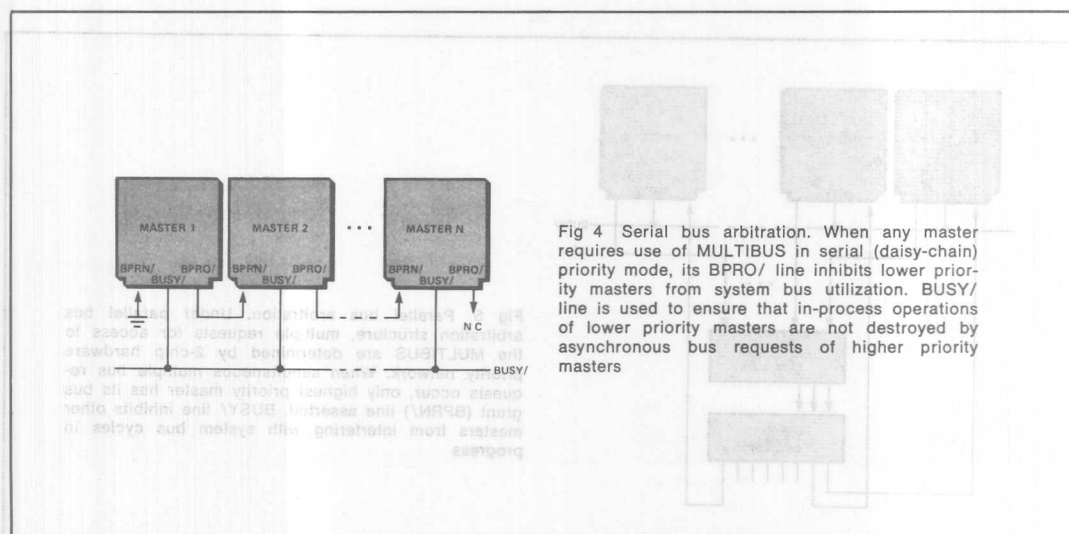


Fig 4 Serial bus arbitration. When any master requires use of MULTIBUS in serial (daisy-chain) priority mode, its BPRO/ line inhibits lower priority masters from system bus utilization. BUSY/ line is used to ensure that in-process operations of lower priority masters are not destroyed by asynchronous bus requests of higher priority masters

such as high speed memory or I/O data transfers and critical read-modify-write operations. With $BUSY/$ asserted in this way, all other masters will find the bus "in use" when they attempt to access it. Whereas system bus transfers normally take place on an interleaved basis (bus arbitration performed for each cycle), this bus lock function permits fast multiple-word transfers, when needed.

Two basic parameters determine the number of masters that can coexist on the system bus in serial bus arbitration mode. These are the $BCLK/$ cycle time and the $BPRN/$ to $BPRO/$ propagation delay of bus masters. Masters may be added to a system as long as the cumulative $BPRN/$ to $BPRO/$ propagation delay is such that the lowest priority master will always have its $BPRN/$ line driven inactive before the next $BCLK/$ falling edge after the highest priority master requests the bus. This worst-case timing condition is met as long as the following relationship is satisfied.

$$\sum_{i=1}^{N-1} (t_{BPRN-BPRO})_i \leq t_{BCLK} - t_{sh}$$

where

$(t_{BPRN-BPRO})_i$ = Propagation delay for master i
 t_{BCLK} = Bus clock ($BCLK/$) cycle time (period)
 t_{sh} = Allowance for bus setup and hold times
 N = Number of bus masters

Using serial bus arbitration and SBC 80 onboard clocks, up to three masters may coexist on the system bus. This number can easily be extended, if desired, by generating a $BCLK/$ with a longer cycle. The SBC 80/05 and 80/20 provide a jumper option which allows the onboard $BCLK/$ to be disabled. This allows the system designer to generate $BCLK/$ externally.

Parallel (Hardware-Encoded) Bus Arbitration

The parallel bus arbitration technique resolves system bus master priorities using external hardware. The

parallel multimaster control line ($BREQ/$) comes into force in this case. Each master asserts $BREQ/$ when it requires access to the system bus. These lines are fed to a 2-chip parallel priority network. As with serial priority resolution, $BPRN/$ acts as the bus access enable input to each master. As Fig 5 illustrates, up to eight master priority levels are encoded by a 74148 priority encoder to a 3-bit code representing the highest priority master currently requesting the system bus. This code drives the 8205 3-to-8 decoder which asserts the proper $BPRN/$ line low to grant bus access to the highest priority master. The 74148/8205 propagation delay is less than 40 ns, easily fast enough to allow eight masters to coexist in this configuration utilizing a $BCLK/$ with a 100-ns period.

Systems requiring up to 16 masters may implement bus arbitration by utilizing two 74148 priority encoders and two 8205 decoders to provide a 16-level hardware priority network. The actual number of bus masters feasible on the system bus will also depend on bus drive/loading considerations. Even under this consideration, systems containing up to 16 masters are feasible.

Thus, single-board computer masters, in conjunction with the MULTIBUS control structure, provide off-the-shelf hardware solutions for the development of efficient multiple processor microcomputer systems. In addition to this hardware capability, the system designer needs to consider several software design issues.

Software Considerations

Several software operations, such as mutual exclusion, communication, and synchronization, are essential to proper multiple processor system operation. The MULTIBUS/SBC 80 functions that enable these software operations are examined.

Mutual Exclusion

In a multiple processor microcomputer system, there are

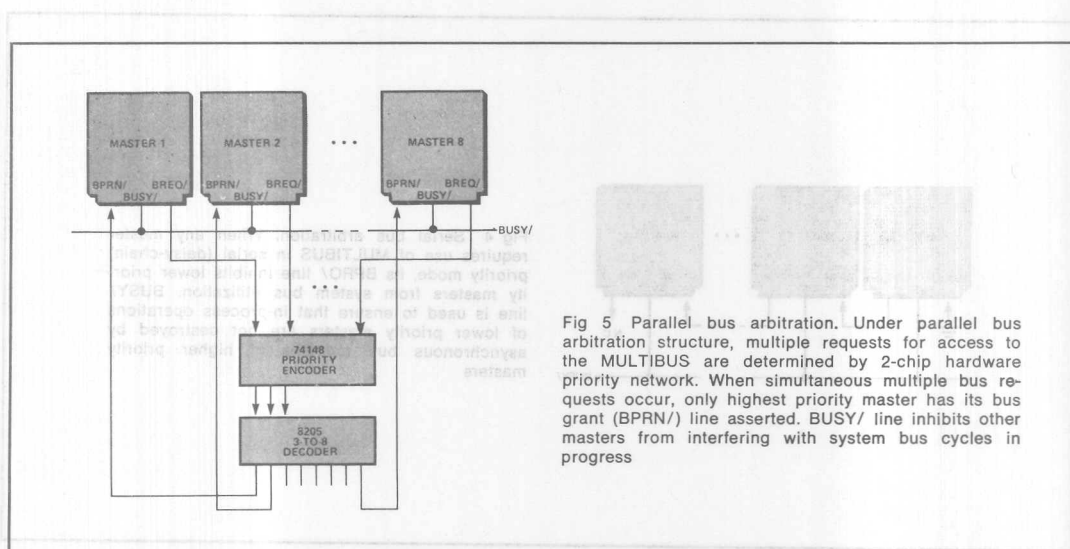


Fig 5 Parallel bus arbitration. Under parallel bus arbitration structure, multiple requests for access to the MULTIBUS are determined by 2-chip hardware priority network. When simultaneous multiple bus requests occur, only highest priority master has its bus grant ($BPRN/$) line asserted. $BUSY/$ line inhibits other masters from interfering with system bus cycles in progress

usually many resources that are shared by the processors. Such shared resources include common memory and peripherals. A properly functioning system must provide a mechanism to guarantee that asynchronous access to those resources is controlled in order to protect data from simultaneous change by two or more processors. Thus, some form of mutual exclusion must be provided to enable one processor to lock out access of a shared resource by other processors when it is in a critical section. A critical section is a code segment that once begun must complete execution before it, or another critical section that accesses the same shared resource, can be executed.

A Boolean variable can be used to indicate whether a processor is currently in a particular critical section (true) or not (false). Testing and setting this variable also presents a critical section. This function must be performed as a single indivisible operation; if it is not, two or more processors may test the variable simultaneously and then each set it, allowing them to enter the critical section at the same time. Such simultaneous entry would destroy the integrity of data and control parameters in global memory or cause erroneous double initialization of a global peripheral controller.

Mutual exclusion can be implemented as a software function alone, as described by Dijkstra⁴, for n processors operating in parallel. The SBC 80/05 and 80/20 bus lock function mentioned earlier provides a means for using program control to simplify mutual exclusion. While the system bus is locked, the master can perform the indivisible test and set operation on the Boolean

variable used to control access to a critical section without intervention from other masters.

Communication

Communication is an essential function that allows a program executing on one processor to send or receive data from a program executing on another processor. Typically, two processors communicate through buffer storage in common memory. One program, called a producer, adds data to buffer storage; another, called a consumer, removes information from buffer storage.

In a typical application, one master may produce buffers of data that are to be consumed by a program executing on another master that services an output device. Communication through buffer storage requires the operations of adding to and taking from buffers. These operations constitute critical sections that can be controlled by providing mutual exclusion around the buffer manipulation operations.

Synchronization

At times there is a need for one master to send a synchronization signal to another. In a sense, synchronization is a special case of communication during which no data is transferred. Rather, the act of signaling is used to "wake up" a program executing on another master. A program may "sleep," by waiting for a synchronizing signal, until it receives a wake-up signal that enables it to continue execution. Manipulation of synchronization signals requires mutual exclusion.

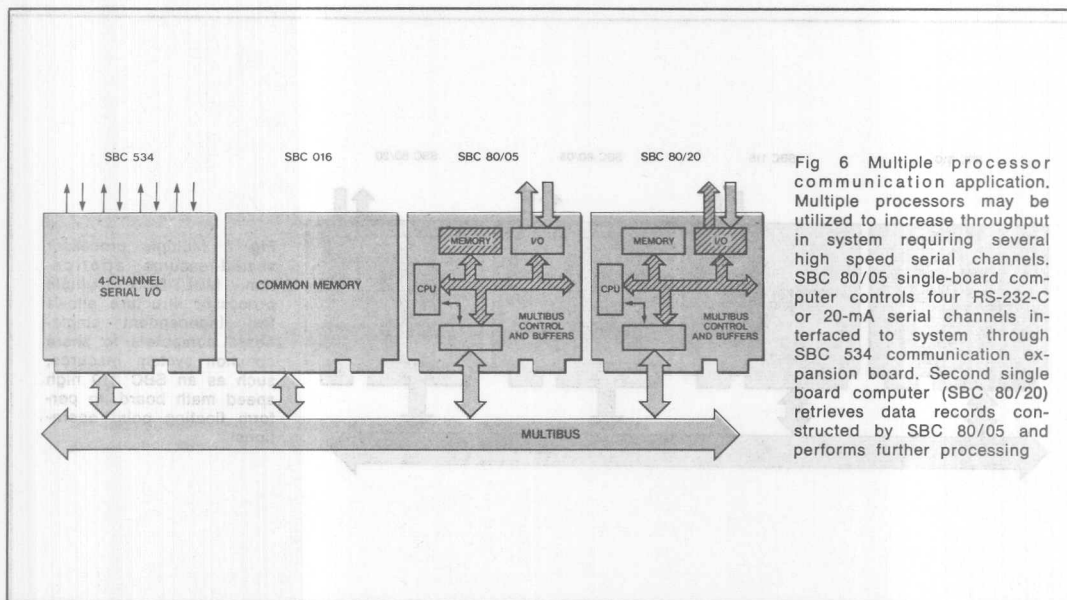


Fig 6 Multiple processor communication application. Multiple processors may be utilized to increase throughput in system requiring several high speed serial channels. SBC 80/05 single-board computer controls four RS-232-C or 20-mA serial channels interfaced to system through SBC 534 communication expansion board. Second single board computer (SBC 80/20) retrieves data records constructed by SBC 80/05 and performs further processing

System Initialization

In a microcomputer system that has multiple processors sharing a common system bus, a system initialization mechanism must be designed to set up the variables that control access to the shared resources. All single-board computers on the MULTIBUS begin execution simultaneously following a system reset. The bus lock function of the computers can be used by one specifically designated master to lock the bus immediately upon system reset and to perform system initialization for common resources before any other master attempts to access them. Since a locked bus has no effect on a single-board computer that is executing out of its local memory and using its local I/O, normal initialization by each processor can proceed while the shared resource initialization takes place.

Multiprocessor Applications

Two applications that are well-suited to multiple processor microcomputer systems are examined. The first provides increased throughput, and the second allows shared resources.

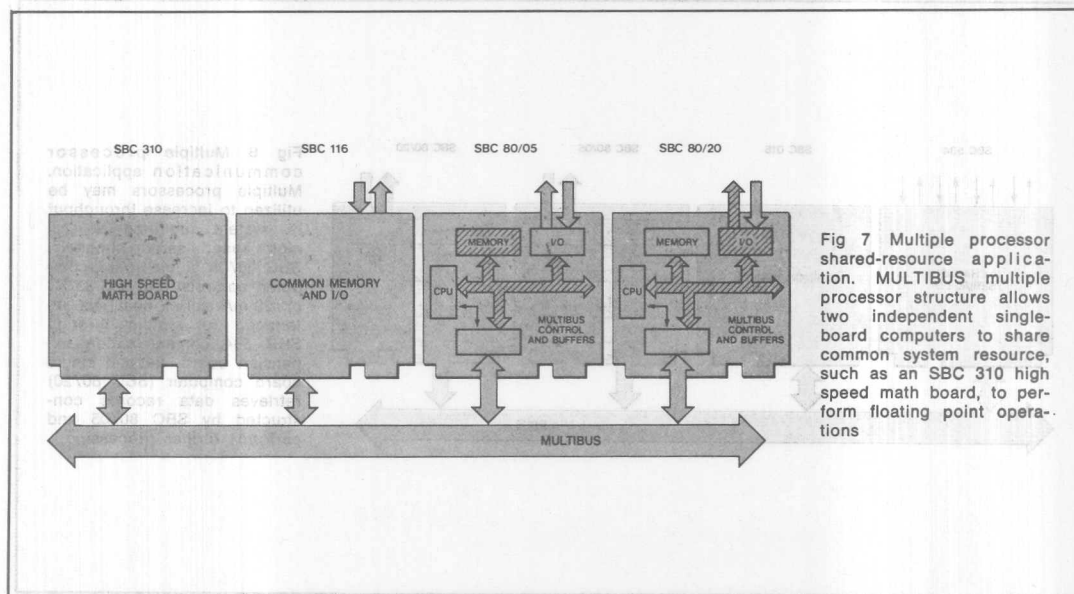
Increased Throughput

Consider a system that is controlling multiple high speed

serial communication channels in addition to other data processing activities. In this case, multiple processors may be utilized to increase system throughput. Such a system with four full-duplex serial channels operating at 4800 baud could produce interrupts every 250 μ s. Interrupts at that frequency in a single master system would leave little time for other processing activities. In a multiple processor approach, one processor can be used to handle the interrupts from the serial channels, accumulate data into records, and then provide those records to another processor by placing them in common memory. The second processor is not burdened with the overhead of handling each character on an interrupt-driven basis, instead it is sent entire records of data available for further processing.

As shown in Fig 6, this application can be handled on the MULTIBUS with four boards. The SBC 80/05 single-board computer is used to service the communication board and prepare the data records. A 4-channel serial communication board (SBC 534) is used to provide the hardware interface for four serial communication channels. The SBC 80/20 single-board computer is used to process data records prepared by the SBC 80/05. Common memory is provided by the SBC 016 16k random-access memory (RAM).

Application of multiple processors to this problem requires communication through buffer storage. Two primitive operations, introduced by Dijkstra⁴, can be used to simplify the communication and synchronization between the masters. These primitives, designated P and V, operate on non-negative integer variables called



semaphores. The V procedure increments the semaphore (S) in a single indivisible operation. To make certain that fetch, increment, and store are not interrupted by another processor, the bus is locked during the operation.

Procedures for P and V primitive operations can be implemented in PL/M⁶ as follows:

```
V:
PROCEDURE (S$ADR);
DECLARE S BASED S$ADR BYTE;
OUTPUT(BUS$LOCK) = LOCK; /* Lock MULTIBUS */
S = S+1; /* Increment semaphore */
OUTPUT(BUS$LOCK) = UNLOCK; /* Unlock MULTIBUS */
END V;
```

The P procedure loops in a busy wait until S is greater than zero, at which time it decrements S. The act of fetching, testing, decrementing, and storing S is also an indivisible operation. Note that if several masters with different speeds are in a busy wait on the same semaphore, the solution presented may not be "fair" to the lower speed processor; that is, the lower speed processor would test the semaphore less frequently, resulting in an unfair advantage for higher speed processors.

Implementation of a procedure for the P primitive is shown in the following PL/M code.

```
P:
PROCEDURE (S$ADR);
DECLARE S BASED S$ADR BYTE;
DO FOREVER:
IF S > 0 THEN /* Test semaphore */
DO:
OUTPUT(BUS$LOCK) = LOCK; /* Lock MULTIBUS */
/* Retest semaphore */
IF S > 0 THEN
DO:
S = S-1; /* Decrement semaphore */
OUTPUT(BUS$LOCK) = UNLOCK; /* Unlock MULTIBUS */
RETURN; /* Exit from P procedure */
END;
OUTPUT(BUS$LOCK) = UNLOCK; /* Unlock MULTIBUS */
/* and continue testing */
END;
END P;
```

It is important to observe in the program listing that S is tested prior to issuing a bus lock. This initial test avoids continuous locking and unlocking of the system bus while looping in a busy wait. The second test is required because another processor could also have found S greater than zero and tried to enter the critical section at the same time.

With the P and V operations, semaphores can be used as resource counters in the buffer manipulation required for communication between the SBC 80/05 and 80/20. For example, a consumer program can use the P operation to decrement the number of full buffers and a V operation to increment the number of empty buffers. In a similar fashion, a producer program can use the P operation to decrement the number of empty buffers and a V operation to increment the number of full buffers. In addition to full and empty buffer counters, it is necessary to maintain linked lists pointing to actual full and empty buffers. A semaphore can be used to provide mutual exclusion around the manipulation of the linked lists. In the example that follows, three variables (FULL, EMPTY, and SEMA) are used to implement these functions. The two PL/M programs illustrate consumer and producer code segments, respectively. Note that the consumer performs initialization because it accesses the semaphores prior to the producer.

```
CONSUMER:
DECLARE EMPTY BYTE EXTERNAL; /* Number of empty buffers */
FULL BYTE EXTERNAL; /* Number of full buffers */
SEMA BYTE EXTERNAL; /* Binary semaphore */
OUTPUT(BUS$LOCK) = LOCK; /* Lock MULTIBUS */
EMPTY = NUMB$BUFFERS; /* Initialize semaphores */
FULL = 0;
SEMA = 1;
OUTPUT(BUS$LOCK) = UNLOCK; /* Unlock MULTIBUS */
DO FOREVER:
CALL P(FULL); /* Decrement full buffer */
/* semaphore */
/* Decrement mutual exclusion */
/* semaphore */
...
...
(Take data from buffer and
place it in local memory,
move buffer from full to
empty linked list)
...
...
CALL V(SEMA); /* Increment mutual exclusion */
CALL V(EMPTY); /* semaphore */
/* Increment empty buffer */
/* semaphore */
...
...
(Process the data)
...
...
END;
END CONSUMER;
PRODUCER:
DECLARE (EMPTY, FULL, SEMA) BYTE EXTERNAL;
DO FOREVER:
...
...
(Prepare data in local
memory)
...
...
CALL P(EMPTY); /* Decrement empty buffer semaphore */
CALL P(SEMA); /* Decrement mutual exclusion */
/* semaphore */
...
...
(Place data in a buffer,
move buffer from empty
to full linked list)
...
...
CALL V(SEMA); /* Increment mutual exclusion */
CALL V(FULL); /* semaphore */
/* Increment full buffer semaphore */
...
...
END;
END PRODUCER;
```

Shared Resources

Another typical application for a multiple processor microcomputer system would be to allow sharing of a resource by two processors. For example, consider two independent processors that have a need for high speed mathematical functions. Although it may not be possible to justify a high speed math module for each system, such a module might be justified if it were to be shared by both processors. A multiple processor microcomputer system could provide the capability to allow both processors to share the math module and not interfere with their otherwise unrelated functions.

This application (illustrated in Fig 7) could be handled with four boards. The SBC 80/05 single-board computer is used to perform various data processing functions requiring high speed floating-point arithmetic. The SBC 80/20 single-board computer controls a process where high speed numeric computations are required. High speed floating-point mathematics functions for both single-board computers are performed by an SBC 310 high speed math unit. SBC 116 combination memory and I/O board provides 16k RAM, 8k electrically programmable read-only memory (EPROM), 48 parallel I/O lines, and an RS-232-C serial port.

The problem to be solved in this application is to ensure that only one processor has access to the shared math module resource at one time. Thus, mutual exclusion must be provided to control the access to the resource. The following PL/M function returns TRUE if access to a critical section, used to implement the mutual exclusion, has been granted.

```

ENTER$CRITICAL$SECTION:
  PROCEDURE (FLAG$ADR) BYTE;
  DECLARE FLAG BASED FLAG$ADR BYTE;
  DECLARE ACCESS BYTE;
  IF FLAG = BUSY THEN
    RETURN FALSE; /* Test flag */
  /* Return false if busy */
  ACCESS = FALSE;
  OUTPUT (BUS$LOCK) = LOCK; /* Lock MULTIBUS */
  IF FLAG = NOT BUSY THEN /* Retest flag */
    DO:
      FLAG = BUSY; /* Set flag busy */
      ACCESS = TRUE; /* and access TRUE */
    END;
    OUTPUT (BUS$LOCK) = UNLOCK; /* Unlock MULTIBUS */
    RETURN ACCESS; /* Return either TRUE or */
    /* FALSE access */
  END ENTER$CRITICAL$SECTION;

```

This PL/M function first tests the flag for the busy condition before issuing a busy lock. As in the P procedure described earlier, this initial test avoids continuous locking and unlocking of the MULTIBUS while a busy wait is being executed. The following procedure performs a busy wait operation on the flag used to control access to a critical section.

```

BUSY$WAIT:
  PROCEDURE (FLAG$ADR) BYTE;
  DO WHILE NOT ENTER$CRITICAL$SECTION (FLAG$ADR);
  END;
END BUSY$WAIT;

```

Typical code segments illustrating the use of these procedures follow.

```

DECLARE MATH$BD$FLAG BOOLEAN EXTERNAL; /* Flag must be */
/* initialized */
MATH$BD$FLAG = NOT BUSY;
...
CALL BUSY$WAIT (MATH$BD$FLAG); /* Here we wait until */
/* we have access */
...
(Process math functions)

```

```

MATH$BD$FLAG = NOT BUSY; /* Set flag not busy */
...
/* We could also test and then do some other */
/* processing if the math module is busy */
IF ENTER$CRITICAL$SECTION (MATH$BD$FLAG)
  THEN DO:
    ...
    (Process math functions)
  ELSE DO:
    ...
    (Something else)
  END;

```

Conclusions

The motivations for implementing multiple processor microcomputer systems include enhanced performance

and throughput. When the appropriate hardware/software design considerations are made, modularity is easily achieved. Hardware solutions to many problems are provided by means of a MULTIBUS structure and SBC 80 single-board computers that have multimaster capability. Through control of MULTIBUS functions, the software designer can perform multiple processor communication, synchronization, and mutual exclusion.

Even with these significant steps toward the simplification of multiple processor microcomputer systems, the design of such systems remains a complex software/hardware design task. The future trend of multiple processor microcomputer systems will be to simplify the software tasks of implementing communications, synchronization, and mutual exclusion. These functions could be performed in varying degrees by additional hardware bus functions.

Potential rewards for a multiple processor architecture include enhanced system throughput, improved real-time response, modular system expansion, and improved system reliability. These benefits will pressure the technology of parallel processing to include microcomputers in an increasing number of computer applications.

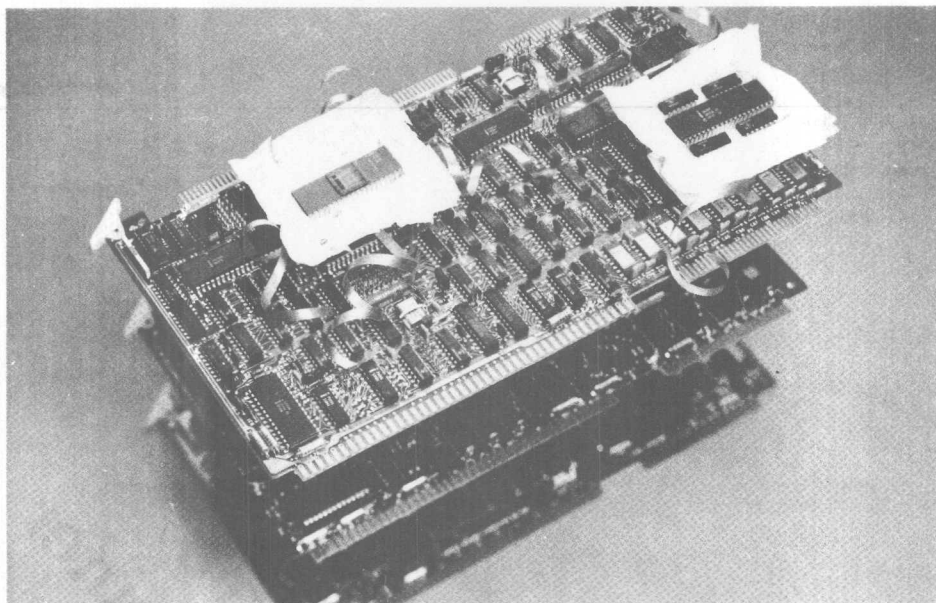
References

1. "SBC 80/05 Hardware Reference Manual," Pub 9800483, Intel Corp, Santa Clara, Calif, 1977
2. "SBC 80/20 Hardware Reference Manual," Pub 9800317, Intel Corp, Santa Clara, Calif, 1976
3. A. C. Shaw, *The Logical Design of Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1974, pp 59-78
4. E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM*, Sept 1965, p 569
5. "Intel MULTIBUS Interfacing," Pub AP-28, Intel Corp, Santa Clara, Calif, 1977
6. D. McCracken, *A Guide to PL/M Programming for Microcomputer Applications*, Addison-Wesley, Reading, Mass, 1978

George Adams, as product line manager for single-chip microcomputers with Intel, is responsible for marketing and applications engineering for MCS-48™ microcomputers. His experience includes work as a microcomputer applications specialist and product planner, and as a computer design engineer. He holds a BSEE from the University of Miami and an MBA from Boston University.

Thomas Rolander is currently a partner of Dharma Systems, a computer systems consulting firm, where he is involved in systems engineering, software, and hardware design. Previously he served as an applications engineering manager for OEM computer systems at Intel. He received a Bachelor's degree in civil engineering and a Master's degree in electrical engineering from the University of Washington.

Triple-bus architecture lets a single-board microcomputer's CPU operate at full speed while other system components share the main memory.



The introduction of Intel's iSBC 80/30 marks the beginning of the third generation of single board computer architecture. Two features separate the new microcomputer from second-generation single-board μ Cs. The major one is a triple-bus architecture that supports a dual-port memory. As a result, the on-board CPU does not tie up the main system bus (Intel's Multibus) when using the memory. Moreover, with two ports, the memory becomes a global resource, accessible via the three buses from the on-board 8085A CPU as well as from remote CPUs and other external devices in multimaster schemes.

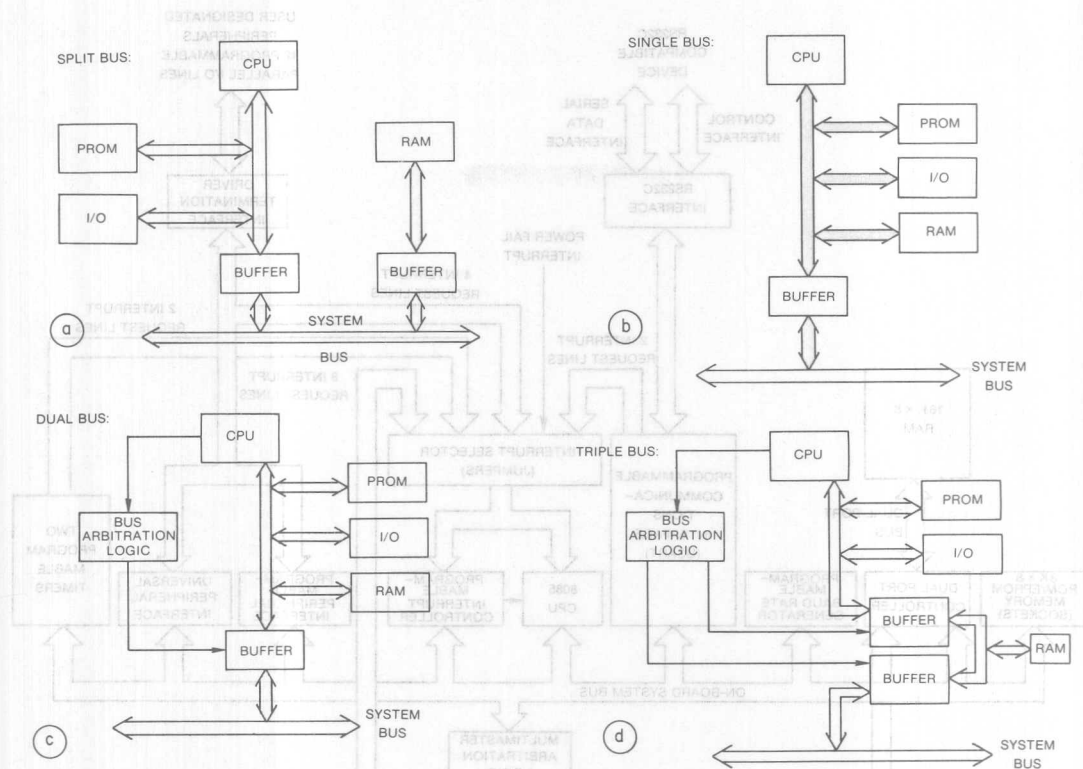
In addition, the 80/30 contains two microprocessors: an 8085A acting as the master CPU and an 8041 single-chip microprocessor acting as a slave, or intelligent-I/O, processor.

Jim Johnson, Project Leader, **Craig Kinnie**, Project Manager, and **Mike Maerz**, Marketing Manager, Intel Corp., Santa Clara, CA 95051.

To appreciate the benefits of the 80/30's triple-bus, dual-port memory architecture, examine the following problem. Now that fully one fourth (16-kbytes) of the available memory space in a 64-kbyte μ C system can reside on a single-board μ C, the CPU must share these 16-kbytes with other system components, such as direct-memory-access devices, discs and other processors. What's the best solution—especially when, in many applications, 16-kbytes is all the memory that's required by the whole system?

Alternatives have problems

The most straightforward way is a split-bus architecture, in which both the CPU and the system have equal access to the memory (Fig. 1a). While the system bus will be able to handle memory access efficiently from devices tied to it, it will be tied up by the CPU—so external operations not related to memory accesses will be hindered.



1. **Microcomputer-bus organizations** takes several forms: In a split-bus approach (a) the CPU and system have equal access to memory, but the CPU ties up the system bus; in a single-bus (b), the CPU encounters extra delays in

using the system bus. A dual-bus structure (c) also has buffer delays, and no system access to on-board memory. But a triple-bus (d) avoids all these problems, allowing total system access to memory.

A single-bus approach (Fig. 1b) is hampered by buffer and system-intervention delays which limit the CPU's performance. And dual-bus architecture (Fig. 1c), while granting the CPU exclusive access, does not allow other bus masters access to the memory. Also dual-bus suffers from buffer delays.

A triple-bus, dual-port architecture (Fig. 1d) provides the benefit of both single and dual-bus architectures: total system access and exclusive access by the CPU. But it also has its disadvantages: Dual-port architecture requires many buffers as well as access-arbitration logic. However, 20-pin octal buffers introduced by several manufacturers don't take up nearly as much board space or cost as much as equivalent standard buffers. Since the octal buffers come in unidirectional or bidirectional forms—and at nearly the same cost—the three-bus approach used on the 80/30 actually takes only as many packages as the split-bus approach.

Access arbitration is solved in the 80/30 with cycle status signals from the 8085A CPU. Instead of providing equal access to the RAM from both the CPU and the system, the arbitration logic is designed to favor the CPU. By assigning the default state of the arbiter

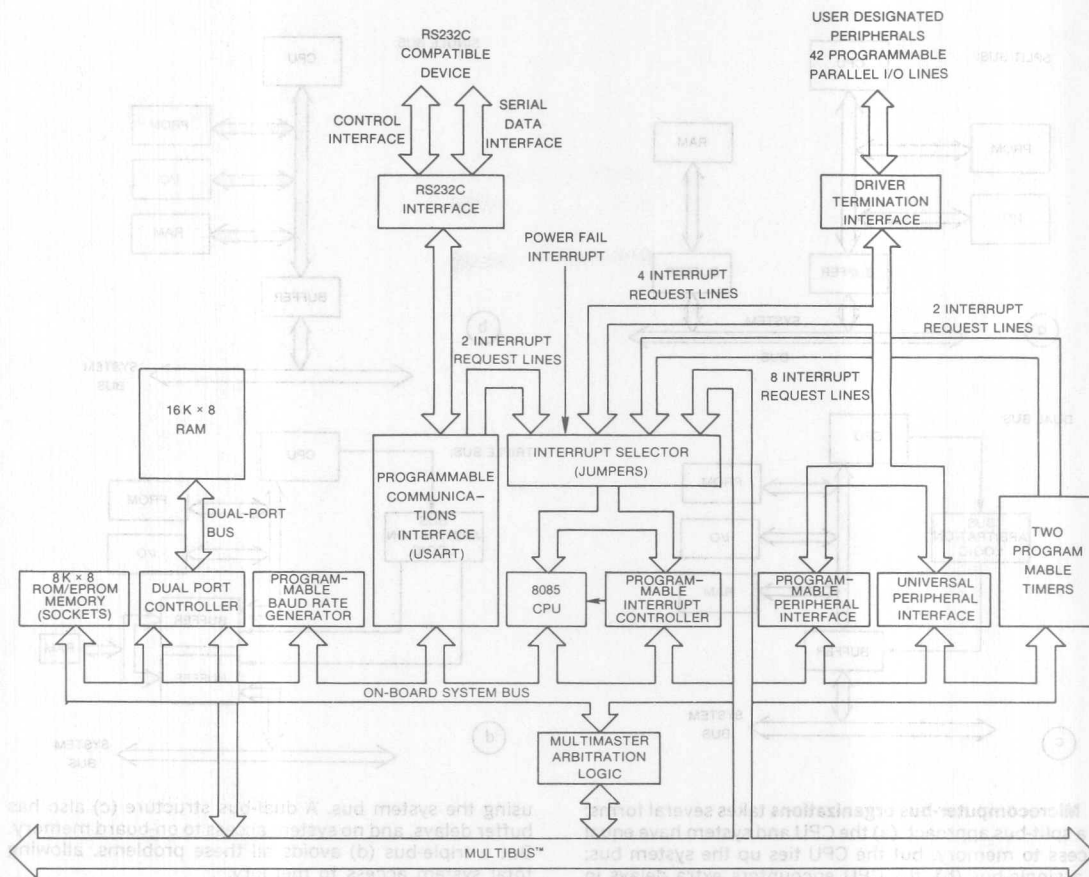
to the CPU, the logic anticipates a CPU memory access and reserves the memory until the cycle is complete.

In addition, if an on-board CPU access is imminent, a reservation signal derived from the 8085A CPU status signals, the ALE (address latch enable), the address, and the cycle status signals (SO, SI, IO/M) will hold off bus contention. As a result, the CPU can operate at full speed without tying up the system bus.

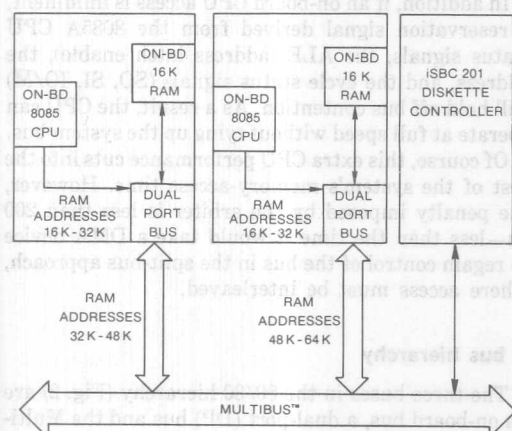
Of course, this extra CPU performance cuts into the rest of the system's memory-access time. However, the penalty imposed by the arbiter is less than 200 ns—less than the time it would take a DMA device to regain control of the bus in the split-bus approach, where access must be interleaved.

A bus hierarchy

The three buses in the 80/30 hierarchy (Fig. 2) are an on-board bus, a dual-port (DP) bus and the Multi-bus (system bus). Innermost is the on-board bus, which connects the 8085A, all on-board I/O peripherals and ROM. The next bus in the hierarchy, the dual-port, connects a dual-port controller, 16-kbytes of dynamic RAM and a dynamic RAM controller. The



2. The full 80/30 one-board microcomputer is organized around its three buses: on board, dual-port, and the external-system Multibus. The main CPU, an 8085A, runs



3. The microcomputer's on-board memory may be addressed independently by the on-board central processor and Multibus bus masters to increase the efficiency of usage of the total available memory space.

at 2.76 MHz, while an 8041A one-chip microprocessor serves as a peripheral controller or slave processor, running with a 2.6-ms cycle time.

outermost bus, the Multibus, offers modules that permit either the expansion or addition of system resources.

With the on-board bus, the 8085A communicates with its on-board I/O and ROM (or PROM, if desirable) and the dual-port bus. Since the on-board bus permits access to the I/O and ROM only from the 8085A, all I/O and ROM (up to 8-kbytes are the 8085A's private property). And as a result, the 80/30 can operate on its on-board bus while another Multibus master uses the Multibus, accessing data from the board's dual-port RAM without reducing processor speed.

The dual-port (DP) bus contains 16-k of read/write memory, implemented with Intel's 2117 16-kbyte dynamic RAM and the 8202 dynamic RAM controller (DRC). The DRC interfaces the DP bus to the 16-kbytes of dynamic RAM, and provides an almost static-RAM type interface. It provides the system with multiplexed addresses, address strobes, and refresh control to the RAM, as well as refresh/access arbitration and acknowledgements.

The RAM on this bus can be accessed from either

the 8085A on the 80/30 or the Multibus. The DP controller arbitrates the RAM requests and performs the bus exchanges.

The DP controller always leaves the DP bus under the control of the 8085A when it is not in use. This permits the 8085A to operate at maximum processor speed when controlling the bus, since there isn't any bus-exchange overhead. When the Multibus requests access to the DP RAM, the DP controller transfers control of the DP bus to the multibus, as soon as the DP bus is not busy. Once the Multibus transfer is complete, the DP bus is returned to the 8085A.

Multiple communication

The DP controller has two independent address decoders—one for decoding Multibus requests, the other for 80/30 requests. This not only permits the address space of the memory to be located in two different parts of memory (Fig. 3), it enables several 80/30s to talk to each other over the Multibus, while sharing the same on-board address as seen by the 8085A. Thus, one program can be loaded in any 80/30 without relinking and relocating the software for execution.

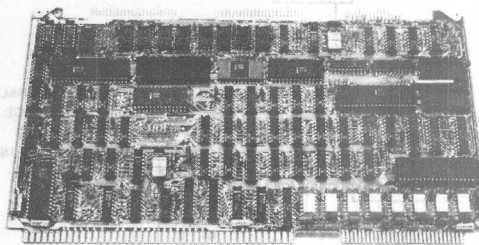
Each bus can communicate either within itself, or with the adjacent bus. Thus, the on-board bus cannot communicate directly with the multibus. However, when the CPU makes a bus request, the on-board and dual-port buses simultaneously determine if they can fulfill it. If the on-board bus can acknowledge the request, it does so, and the DP bus control is not required to determine if the DP bus can acknowledge the request. If the DP bus, not the on-board, can acknowledge the request, it does so, and the controller then lets the CPU use the bus. Thereafter, the RAM controller completes the operation and generates an acknowledge signal.

If neither the on-board nor DP bus can fill the bill, the Multibus is solicited by the CPU. Since a bus can only communicate with an adjacent bus, the on-board bus must request the DP bus to communicate with the Multibus via the DP controller. The on-board bus will retake control of the DP bus only after the request to use the Multibus is granted. This prevents lockout problems with the DP bus, where the CPU requests the Multibus when it is controlled by another bus master accessing the DP RAM.

How the 80/30 performs is directly related to how many buses it must use to complete a requested operation. The on-board bus always operates at maximum processor speed. The DP bus operates at maximum only if it hasn't been busy and a memory refresh cycle was not in process. The processor speed when the Multibus is used depends on bus overhead involved and the type of module requested.

The 80/30 boasts more than a three-bus architecture. For one thing, its I/O is designed to interface to a wide variety of external devices, including switches, motor drives, bistable sensors, displays,

The 80/30 in brief



The iSBC 80/30 uses the latest LSI components to obtain the highest performance of any Intel single-board computer. Built on a 6.75×12 -in. board, it contains the following features:

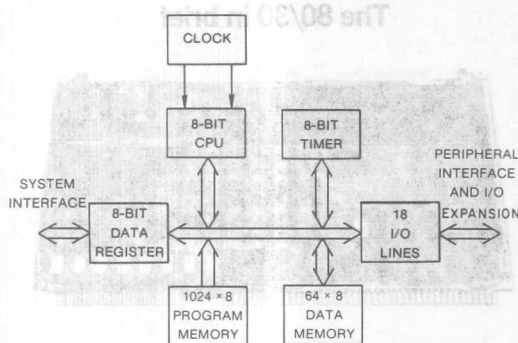
- 8085A central processor operating at 2.76 MHz.
- 16-kbytes of dual-port RAM using Intel's new 16-kbyte dynamic RAMs and 8202 dynamic RAM controller.
- Sockets for 2, 4 or 8-kbytes of ROM using Intel's 2758, 2708, 2716, or 2332 EPROMs or ROM replacements.
- A socket for Intel's 8041A/8741A universal peripheral interface (UPI) having 18 software-configurable I/O lines with sockets for drivers/terminators.
- A programmable serial-communication channel with RS-232 interface and programmable baud rate.
- Multibus control logic which allows up to 16 masters to share the system bus.
- 12 vectored priority interrupts.
- Two programmable 16-bit BCD or binary internal timers.

keyboards, printers, teletypewriters, communicator modems, cassettes and other computers. This versatility is provided with LSI programmable devices such as Intel's 8255 programmable parallel I/O device, 8251A programmable communication channel, 8253 interval timer, 8259 interrupt controller, and 8041A/8741A universal peripheral interface (UPI).

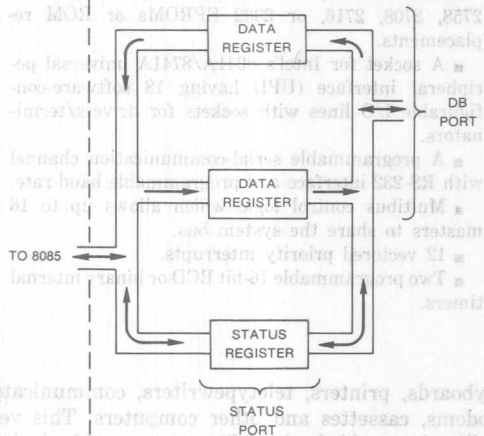
The slave processor

The ability to interface this wide variety of external devices is facilitated by the 8041A/8741A UPI (Fig. 4), which can be added to the 80/30. The UPI is a complete single-chip microcomputer which acts as a peripheral to the 8085A. It is completely user-programmable with 1-kbyte of ROM (8041A) or EPROM (8741A) memory for data storage. The UPI allows you to fully specify your control algorithm in the peripheral chip without relying on the 8085A. Devices such as printer controllers and keyboard scanners can be completely self-contained, relying on the 8085A only for data transfers.

The UPI is a powerful 8-bit CPU with a 2.6-ms cycle time and an instruction set optimized for bit manipu-



4. The 8041A/8741A single-chip microcomputer (UPI-41) has its own on-chip ROM and RAM and can be programmed to perform various peripheral control functions.



5. The UPI's two data registers are organized so that the 8085A CPU can write in just one register and read from the other. As a result, the two registers appear as one register to the main 8085A CPU.

lation and I/O operations. It contains an 8-bit counter/timer, buffers to communicate with the 8085A, and two 8-bit programmable I/O ports, which can be customized by software or by plugging in suitable line drivers or terminators into sockets. The UPI also has two input bits that it can test directly. An RS-232 driver and receiver on the 80/30 permit the UPI to be programmed as a simple serial-communication channel.

Interfacing to the on-board bus

The UPI interfaces asynchronously with the on-board bus using two data and two status registers. The UPI's two internal data registers appear to the

8085A as only one register, since one data register can be written into only by the UPI and read only by the 8085A, and the other can be written into only by the 8085A and read by the UPI (Fig. 5). This is done to prevent the two CPUs from simultaneously writing into a data register.

The UPI can communicate with the 8085A by loading a data register and then returning to its previous control task. The 8085A can periodically poll the UPI status port for the valid-read (VR) flag, which is set in hardware when the UPI writes to its data port, or the UPI can generate an interrupt to the 8085A via an I/O bit that can be programmed to be the VR flag.

Once the 8085A determines the VR flag is true, it can transfer the data to its own memory without disturbing the UPI. The VR flag is automatically cleared after the data are transferred. Similarly, when the 8085A transfers data to the UPI, a valid output (VO) flag is set and an interrupt to the UPI is generated (if enabled) automatically. Once the UPI transfers the data, the VO flag is cleared. The VO flag can also be programmed to a port bit for generating interrupts to the 8085A to indicate that the transfer is complete.

An extensive interrupt system

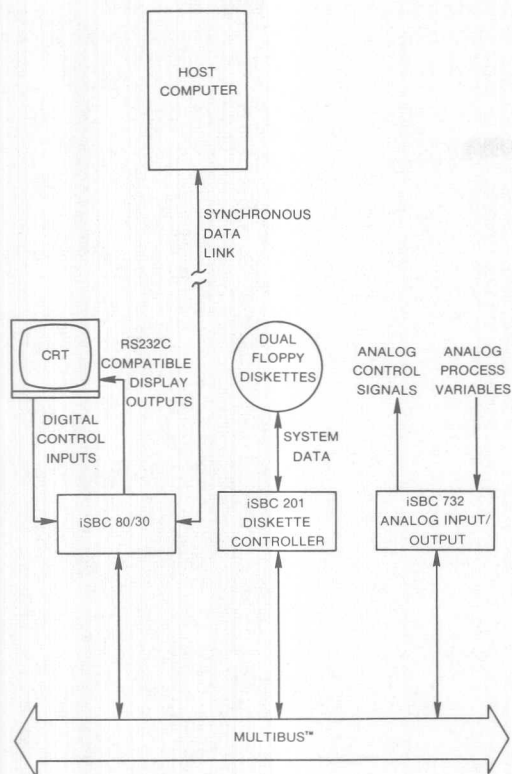
The 80/30 provides 12 vectored priority interrupts, four of which are handled directly by the 8085A's interrupt-processing capability and routed to fixed, unique memory locations. The remaining eight levels are handled via the 8259A programmable interrupt controller (PIC), which generates a unique memory address for each level. These addresses are equally spaced at intervals of four or eight (software-selectable) bytes. This 32 or 64-byte block may be located to begin at any 32 or 64-byte boundary in the 65,536-byte memory space. A single 8085A jump instruction at each of these addresses then provides the linkage to locate each interrupt-service routine independently anywhere in memory. The PIC provides a selection of four priority algorithms so that the manner in which real-time requests are processed may be configured to meet the requirements of the system under design.

The 80/30 also has two 8253-based programmable 16-bit BCD and binary timers/event counters, which can be used for a variety of functions. Both timers may be set to act as a rate generator (divide-by-N counter), a square-wave generator, a programmable retriggerable one-shot, or one of the timers can be jumper-selected as an event counter. In addition, an interrupt can be generated when a time interval has expired or when a specified number of events has occurred.

To see how useful the 80/30 can be, consider a supervisory control/monitoring system (Fig. 6) using an Intel iSBC 80/30 single-board computer, iSBC 201 diskette controller, and iSBC 732 analog input/output

ELECTRONIC DESIGN 15, July 19, 1978

AFN-01931A



6. In this application example, the 80/30 forms the heart of a remote data-acquisition system. By taking advantage of the one-board microcomputer's dual-port memory and universal peripheral interface, the system achieves a combination of attractive cost and efficiency.

board. Here local commands and process-status signals are given and displayed on a CRT, which is interfaced via the iSBC 80/30 resident UPI and RS232C components. Process variables are converted from analog to digital using the analog I/O board. Control variables are passed over the Multibus from the 80/30 to the 732, where they are converted from digital to analog.

System data are logged on two diskettes, which are controlled by the 201. The controller board's on-board DMA interface accesses the 80/30's dual-port memory and stores the data on one of the floppy discs.

At the end of the day, a remote host processor, interfaced to the 80/30 via a modem (through the 80/30's 8251A and RC232C circuits) can request all or part of the diskette-resident data. Here, the 80/30 uses its on-board dual-port memory as a data buffer for transfers to the host.

Intel's RMX/80 real time executive, disc-file system and analog drivers provide the majority of the system's software.■

Note: Multibus and iSBC are registered trademarks of Intel Corp.

On-board random-access memory, accessible from system bus, makes input/output controller subsystem look like just another memory board to the host microprocessor.

by Craig Kinnie and Michael Maerz, Intel Corp., Santa Clara, Calif.

Input/output controllers based on microprocessors throughout microcomputer systems by relieving the host processor of tedious time-consuming control tasks—and a new design concept that increases the processing capability of this subsystem promises to hike throughput even more. It will cut the host intervention needed to transfer data and to run the controller.

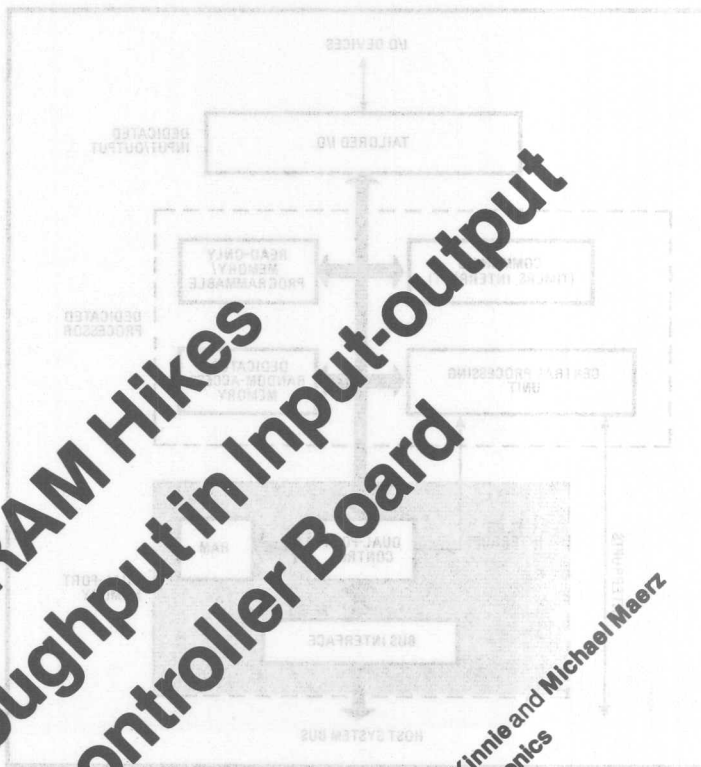
November 1978

Although this concept allows the subsystem to remain dedicated to its I/O control function and to assume a supervisory role to the host processor, it has more processing power than previous generations of such controllers. Hence it has been dubbed the intelligent-slave concept by Intel, which applies it in the iSB-C 244 intelligent communication controller.

The new subsystem architecture is divided into three major sections: dedicated I/O, dedicated computer, and dual-port memory (Fig. 1). The dedicated input/output

section of dual-port memory that resides in the controller subsystem. This setup allows more efficient transfer of large blocks of data from the I/O device to the system bus without contention over access to the system bus. It also

It. Most of memory. New controller architecture includes the dedicated input/output section, and dedicated processor of an intelligent peripheral controller, but its heart is the dual-port random-access memory.



Dual-port RAM Hikes Throughput in Input-output Controller Board

Craig Kinnie and Michael Maerz
Electronics

Reprinted with permission from Electronics
Copyright Motorola, Inc., 1978

Electronics/August 17, 1978

Dual-port RAM hikes throughput in input/output controller board

On-board random-access memory, accessible from system bus, makes input/output controller subsystem look like just another memory board to the host microprocessor

by Craig Kinnie and Michael Maerz, Intel Corp., Santa Clara, Calif.

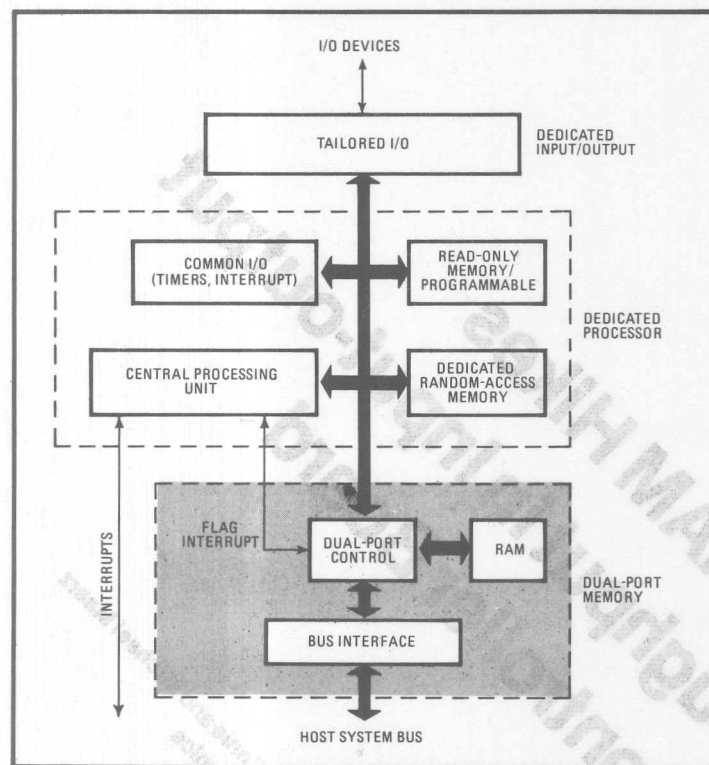
□ Input/output controllers based on microprocessors step up throughput in microcomputer systems by relieving the host processor of tedious, time-consuming control tasks—and a new design concept that increases the processing capability of this subsystem promises to hike throughput even more. It will cut the host intervention needed to transfer data and to run the controller.

In this configuration, all communications between the host processor and the controller are handled through a section of dual-port memory that resides in the controller subsystem. This setup allows more efficient transfer of large blocks of data from the I/O device to the system without contention over access to the system bus. It also

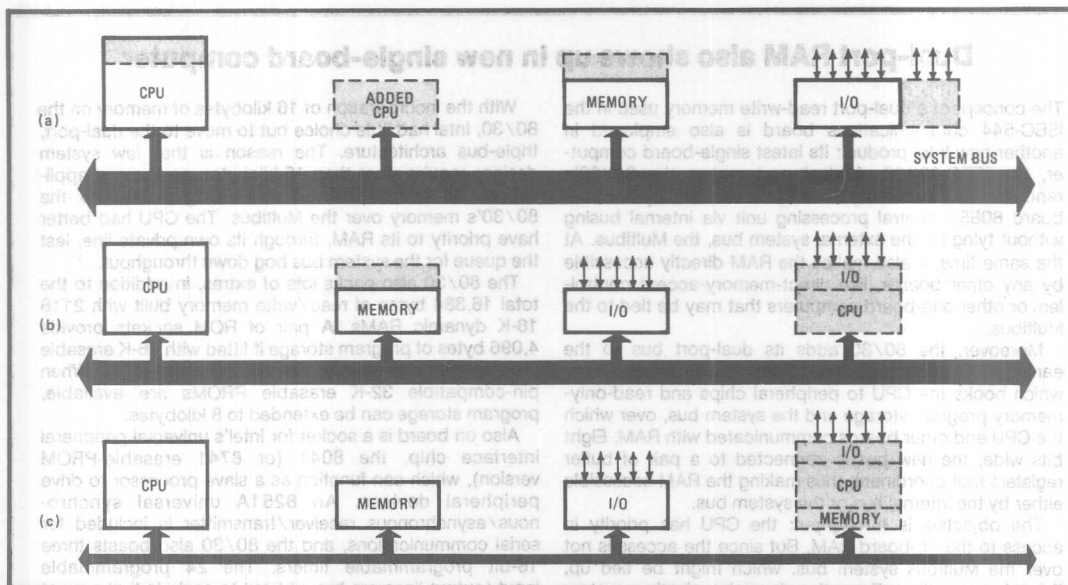
simplifies interprocessor communications because the subsystem controller appears to the host processor simply as an additional RAM board.

Although this concept allows the subsystem to remain dedicated to its I/O control function and to assume a subservient role to the host processor, it has more processing power than previous generations of such controllers. Hence it has been dubbed the intelligent-slave concept by Intel, which applies it in the iSBC 544 intelligent communications controller.

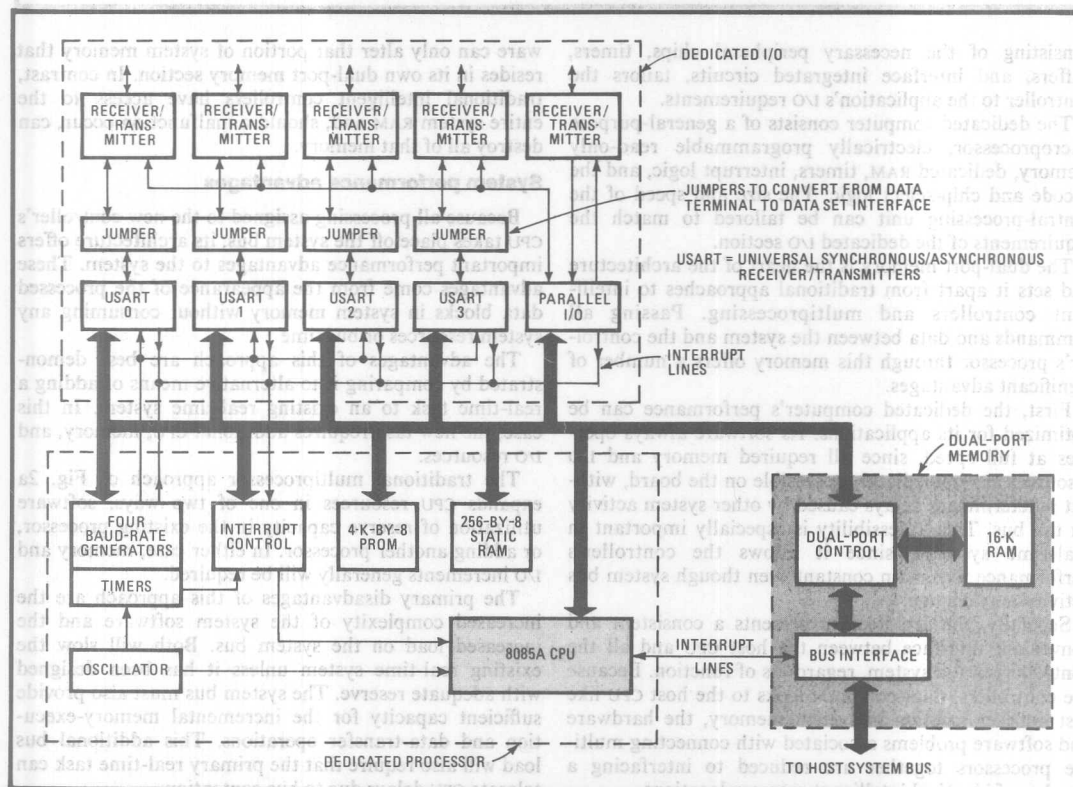
The new subsystem architecture is divided into three major sections: dedicated I/O, dedicated computer, and dual-port memory (Fig. 1). The dedicated input/output,



1. Heart of memory. New controller architecture includes the dedicated input/output circuitry and dedicated processor of an intelligent peripheral controller, but its heart is the dual-port random-access memory.



2. Performance advantages. In adding a real-time task to an existing real-time system, the load on the system bus is significantly reduced over the traditional multitasking approach (a) or the intelligent controller approach (b) by the intelligent-slave controller approach (c).



3. The 544. Based on the 8085A microprocessor with 4 kilobytes of PROM and 16 kilobytes of RAM, the subsystem is designed as a communications controller with four synchronous/asynchronous buffered serial I/O channels, and a 10-bit parallel I/O interface.

Dual-port RAM also shows up in new single-board computer

The concept of a dual-port read-write memory used in the iSBC-544 communications board is also employed in another new Intel product: its latest single-board computer, the iSBC-80/30. A dual port makes the 80/30's random-access memory directly accessible by the on-board 8085A central processing unit via internal busing without tying up the external system bus, the Multibus. At the same time, it also makes the RAM directly accessible by any other boards, like direct-memory-access controllers or other one-board computers that may be tied to the Multibus.

Moreover, the 80/30 adds its dual-port bus to the earlier iSBC computers' pair of buses: an internal bus, which hooks the CPU to peripheral chips and read-only-memory program storage and the system bus, over which the CPU and other boards communicated with RAM. Eight bits wide, the new bus is connected to a pair of buffer registers that coordinate, thus making the RAM accessible either by the internal bus or the system bus.

The objective is throughput: the CPU has priority in access to the on-board RAM. But since the access is not over the Multibus system bus, which might be tied up, there is no waiting. From the viewpoint of other system boards, the system bus is accessible a greater percentage of the time.

With the incorporation of 16 kilobytes of memory on the 80/30, Intel had little choice but to move to the dual-port, triple-bus architecture. The reason is that few system designs require more than 16 kilobytes, so in many applications all boards will be demanding access to the 80/30's memory over the Multibus. The CPU had better have priority to its RAM, through its own private line, lest the queue for the system bus bog down throughput.

The 80/30 also packs lots of extras, in addition to the total 16,384 bytes of read/write memory built with 2116 16-K dynamic RAMs. A pair of ROM sockets provide 4,096 bytes of program storage if fitted with 16-K erasable programmable read-only memories like the 2716. When pin-compatible 32-K erasable PROMs are available, program storage can be extended to 8 kilobytes.

Also on board is a socket for Intel's universal peripheral interface chip, the 8041 (or 8741 erasable-PROM version), which can function as a slave processor to drive peripheral devices. An 8251A universal synchronous/asynchronous receiver/transmitter is included for serial communications, and the 80/30 also boasts three 16-bit programmable timers. The 24 programmable input/output lines are brought out to sockets that accept quad line-drivers or -terminators for interfacing.

Ray Capece

consisting of the necessary peripheral chips, timers, buffers, and interface integrated circuits, tailors the controller to the application's I/O requirements.

The dedicated computer consists of a general-purpose microprocessor, electrically programmable read-only memory, dedicated RAM, timers, interrupt logic, and the decode and chip-select logic. The size and speed of the central-processing unit can be tailored to match the requirements of the dedicated I/O section.

The dual-port memory is the heart of the architecture and sets it apart from traditional approaches to intelligent controllers and multiprocessing. Passing all commands and data between the system and the controller's processor through this memory offers a number of significant advantages.

First, the dedicated computer's performance can be optimized for its applications. Its software always operates at full speed, since all required memory and I/O resources are immediately accessible on the board, without indeterminate delays caused by other system activity on the bus. This accessibility is especially important in real-time systems, since it allows the controller's performance to remain constant even though system bus activity may change.

Secondly, the architecture presents a consistent and convenient interface between the host CPU and all the controllers in the system, regardless of function. Because the controllers' dual-port RAM looks to the host CPU like just another location in system memory, the hardware and software problems associated with connecting multiple processors together are reduced to interfacing a number of identical intelligent memory locations.

Also, the architecture offers a degree of protection for the data in memory. The subsystem computer and soft-

ware can only alter that portion of system memory that resides in its own dual-port memory section. In contrast, traditional intelligent controllers have access to the entire system RAM and, should a malfunction occur, can destroy all of that memory.

System performance advantages

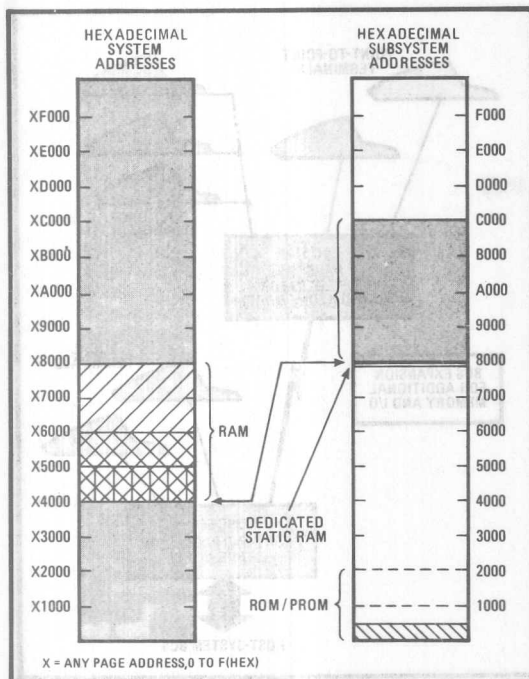
Because all processing assigned to the new controller's CPU takes place off the system bus, its architecture offers important performance advantages to the system. These advantages come from the appearance of the processed data blocks in system memory without consuming any system resources or bus time.

The advantages of this approach are best demonstrated by comparing it to alternative means of adding a real-time task to an existing real-time system. In this case, the new task requires additional CPU, memory, and I/O resources.

The traditional multiprocessor approach of Fig. 2a expands CPU resources in one of two ways: software utilization of reserve capacity in the existing processor, or adding another processor. In either case, memory and I/O increments generally will be required.

The primary disadvantages of this approach are the increased complexity of the system software and the increased load on the system bus. Both will slow the existing real-time system unless it has been designed with adequate reserve. The system bus must also provide sufficient capacity for the incremental memory-execution and data-transfer operations. This additional bus load will also require that the primary real-time task can tolerate CPU delays due to bus contention.

The intelligent-controller approach of Fig. 2b has gained widespread use since the advent of the micropro-



4. Memory mapping. The variable system memory addresses are always mapped into the on-board address of 8000HEX, providing software independence for the subsystem and the host.

processor. This approach combines the CPU and I/O increments onto a single module that usually includes direct-memory-access transfer logic. In some cases the execution memory for the CPU is included.

This approach lessens the bus-loading problem since the I/O data transfers and some memory-execution cycles take place off the system bus. However, both CPUs' programs will have to tolerate delays caused by increased bus contention. Increased software sophistication is the primary disadvantage of this approach, much as with the multiprocessing approach of Fig. 2a.

The intelligent-slave approach of Fig. 2c can be viewed as a logical extension to the intelligent-controller approach. Combining the CPU, I/O, and memory increments creates a single module that has a minimal impact on the existing system software and bus loading. What's more, the subsystem can operate at full capacity without regard to other system activity. It can be programmed outside the primary system and then added with minimal impact on the system software or performance.

A limitation of the approach is the inability of the subsystem to transfer data into portions of the system memory space that reside off its board. This problem is minimized by the ability of the controller's RAM to serve as a substantial portion of the entire memory space addressable by the system. In this light, the on-board processor can be viewed as having a DMA capability limited to a portion of the system's address space.

In a system with more than one of the new controllers, the system CPU handles any data that must be trans-

TABLE: EIA RS-232-C SIGNALS PROVIDED AND SUPPORTED

Carrier detect	Receive clock
Clear to send	Receive data
Data set ready	Ring indicator
Data terminal ready	Transmit clock
Request to send	Transmit data

ferred from one to another. Applications involving the transfer of large blocks of data would be best served by a central block-transfer device elsewhere on the bus.

The advantages offered by the new approach in this example of adding onto an existing system are just as applicable to a ground-up design. This modular approach to configuring real-time multiprocessing systems simplifies hardware and software design, as well as system integration.

While the primary design objective of the new architecture is operation in a multiprocessing system, it can provide significant utility as stand-alone processors. Thus these controllers incorporate a second mode of operation called the limited bus-master mode.

In this mode of operation the new controller can be used like a single-board computer as long as it is the system's only master of the bus. It can be connected to standard memory or I/O expansion boards to enhance its capability. It can even be used to drive other such controllers as long as they are used in the subsystem mode. This dual operational mode will allow the new controllers to serve a broad range of applications.

Communications first

Communications applications present complex processing requirements and an inherent real-time nature, so it is logical that a communications processor be the first of these new controllers to be marketed. The ISBC 544 intelligent communications controller can serve as a flexible front end to an iSBC system or as a cost-effective stand-alone processor configured as a terminal cluster or line concentrator. Its design (Fig. 3) incorporates an 8085A CPU, 16 kilobytes of dual-ported dynamic RAM, 4 kilobytes of PROM, programmable interrupt control, three interval timers, four programmable baud-rate generators, four synchronous/asynchronous buffered serial I/O channels, and a 10-bit parallel interface compatible with a Bell 801 automatic calling unit.

The dual-port memory block basically consists of the 16-kilobyte bank of random-access memory, which is accessible from either the system bus or the on-board processor through the dual-port controller. This memory block provides the primary means of communication between the system and the on-board 8085A. The port to the memory, which looks to the system bus like any other RAM card belonging to the system, features full 20-bit

addressing and a typical access time of 600 nanoseconds.

The interface's address-decode logic allows switching of the base address of the iSBC 544 to any 4-kilobyte boundary in the host system's address space. In addition, the user may reserve 8, 12, or 16 kilobits of the 544's memory for use by the on-board processor only. This reserved memory is not accessible from the system bus and does not occupy any system address space. The only restriction is that all of the unreserved memory reside in the same 64-K address page of the system memory.

This memory division can be a significant advantage in large 8-bit microcomputer systems. Only that portion of the controllers' memory needed to pass data between CPUs must be made accessible to the system. The remaining buffer and execution memory does not consume any system address space.

The net result is an increase in the system's overall memory capacity. For example, a microcomputer system that would usually be limited to 64 kilobytes of memory has a total memory capacity of over 190 kilobytes when driving seven 544s.

Address maps and interrupts

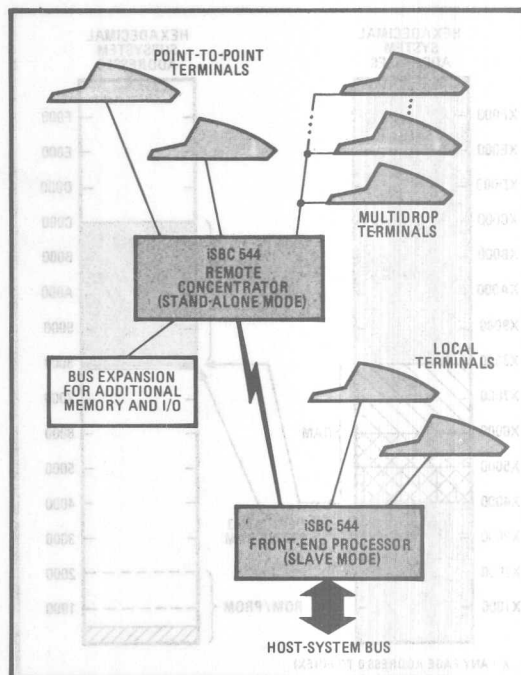
To the on-board processor, the base address of its memory is fixed at 8000HEX. Furthermore, all on-board addresses are fixed, so that multiple 544s operating on the same system bus can be running identical programs regardless of their base address on that bus. This capability necessitated the address-mapping logic to transform addresses from the system bus into the equivalent in the on-board address space starting at location 8000HEX (Fig. 4).

The address-mapping logic also implements the flag-interrupt feature. It provides an interrupt to the on-board processor whenever a byte is written into the 544's base address from the system bus, and a read from the on-board processor to the base address clears the interrupt. Since each 544 in a system has a different base address in that system's RAM, it also has a unique interrupt. This flag-interrupt capability is a key element in establishing a protocol for communications between the host CPU and the subsystems' processors.

The dual-port control logic is responsible for resolving contention over access to the memory and is designed to optimize the performance of the subsystem CPU. Unless the system bus has initiated a memory cycle before the on-board processor requests memory, that CPU runs at full speed. The maximum delay that can be encountered is one memory cycle. The arbitration logic actually reserves the memory for the on-board processor before it generates the necessary commands. This advance reserving guarantees that the on-board CPU will suffer minimum intervention from system bus accesses.

When the iSBC 544 is used in the stand-alone limited bus-master mode, the dual-port logic is disabled and the bus interface buffers are turned around to drive onto the bus. This reversal allows the on-board central-processing unit access to the memory of other subsystems or I/O expansion boards on the system bus.

The dedicated computer is built with an 8085A CPU operating at 2.76 megahertz, between 2 and 4 kilobytes of PROM and ROMs or 8 kilobytes of ROM using 2332



5. Communications applications. Two typical applications of the new iSBC 544 would be as a front-end communications processor to a microcomputer system and as a remote concentrator to a series of point-to-point or multidrop connected terminals.

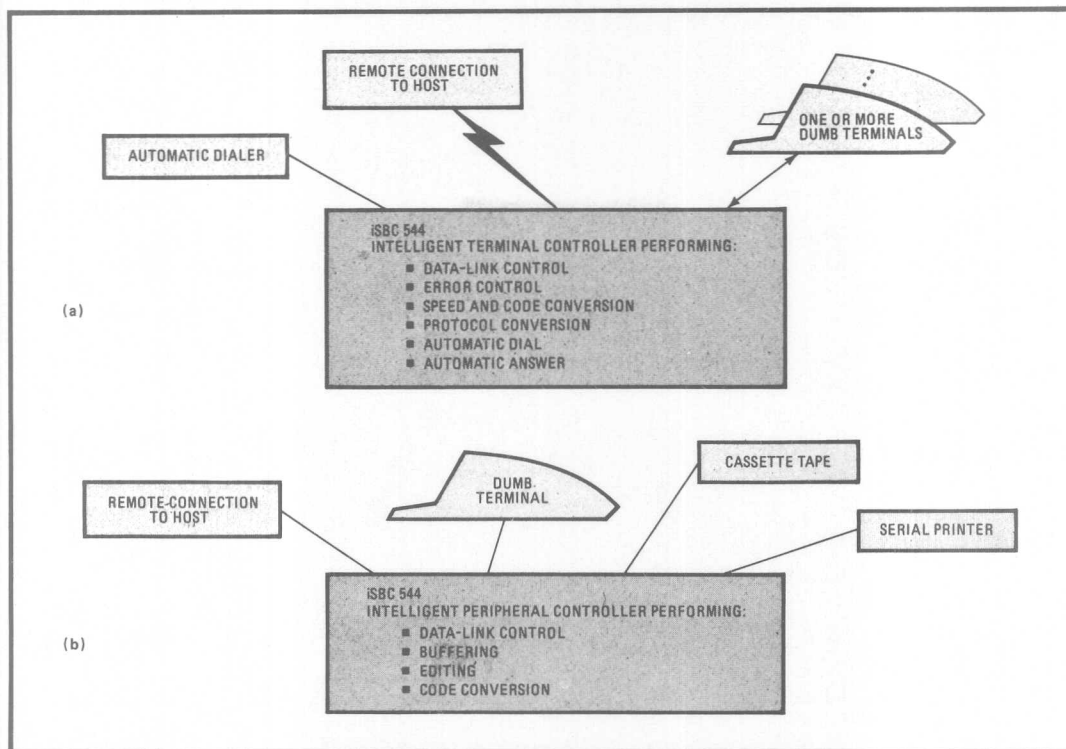
mask-programmable parts, 256 bytes of static RAM, two 16-bit and one 14-bit interval timers, and a 8259 programmable interrupt controller for individual receive or transmit interrupt inputs for each serial port.

Special command-decode logic was added to the CPU to allow it to operate at maximum speed independent of other system activity. There are 21 sources of interrupt on the 544, including the separate transmit and receive interrupts for each port and separate timer interrupts. In addition to receiving an interrupt from the system, the 544 can also send an interrupt to the system bus via the 8085A's serial-output data line.

Since this controller is intended for communications applications, latched interrupts are provided directly to the CPU for loss of carrier and ring indicator for all four I/O ports. The ring-indicator and carrier-detect lines can also be monitored through the parallel port.

Dedicated I/O

The dedicated I/O section of the 544 provides a high degree of flexibility and programmability. This results primarily from the inclusion of four 8251A universal synchronous/asynchronous receiver/transmitters. These devices are programmable for synchronous or asynchronous mode, character size, parity bits, stop bits, and baud rates. Data, clocks, and control lines are buffered with RS-232-C-compatible drivers and receivers to four 26-pin card-edge connectors. Each port is configured as a data-terminal interface, but may be converted to a



6. From slave to master. In its stand-alone mode, the 544 can operate as a bus master and be configured as an intelligent terminal controller connecting dumb terminals to a data link (a) or as a peripheral controller connecting RS-232-C-compatible units to the terminal (b).

data-set interface by changing a single jumper-plug assembly. The ports support most RS-232-C signals (those that are listed in the table).

A programmable baud-rate generator is also provided for each port. The range of baud rates available is 75 to 56 kilobits per second. The generators are implemented with 8253 programmable interval timers, which receive a jumper-selectable input frequency of 1.84 or 1.23 MHz. In addition, one of the CPU's interval timers can be converted to baud-rate operation and jumpered to any port to provide it with split-speed operation.

The 544 also provides a parallel port with four RS-232-C buffered input lines and six RS-232-C buffered output lines. This port is configured to interface to most automatic calling units but may be used as a general-purpose I/O port. It is implemented with an 8155 programmable peripheral interface that also provides the 256 bytes of static RAM and the 14-bit timer.

Applications

A likely common use of the 544 as a subsystem is as a front-end processor or terminal multiplexer (Fig. 5) in an iSBC system. The 544 performs all communications-related functions such as format control, code conversion, data-link control, error checking, data compression, and protocol management. It can handle multiple protocols, line speeds, and data formats.

All the system processor sees are the processed data

blocks that appear in system memory. An automatic dialer could be added to provide a dial-up connection to a host processor or network.

Also shown in Fig. 5 is another 544 used in its limited bus-master mode as a remote concentrator and terminal controller. The line and memory capacity of the remote concentrator can be increased by the addition of standard iSBC memory and I/O expansion boards.

The intelligent-terminal controller shown in Fig. 6a is a prime example of a 544 used in the stand-alone mode. It can connect one or more dumb terminals to a data link and provide the necessary buffering, code conversion, and data-link control. It could also connect a terminal that happens to communicate in a different protocol to a new network or to more than one network.

The iSBC 544's multiple serial lines do not have to be used for communications. They can also be used to connect RS-232-C-compatible peripherals to the terminal (Fig. 6b). In this configuration, the 544 can provide message editing and formatting, bulk storage, and hard-copy output.

As this last application suggests, the 544 is the vanguard of a family of intelligent I/O controllers that will add tremendous increases in throughput and versatility to the iSBC line of single-board computers. The basic architecture will simplify the task of developing multiprocessing hardware and software solutions that will overcome throughput limitations. □

AFN-01931A

16-bit single-board computer maintains 8-bit family ties

Three-bus 8086-based board addresses a megabyte, communicates over expanded system bus

by Robert Garrow, Jim Johnson, and Les Soltesz, Intel Corp., Santa Clara, Calif.

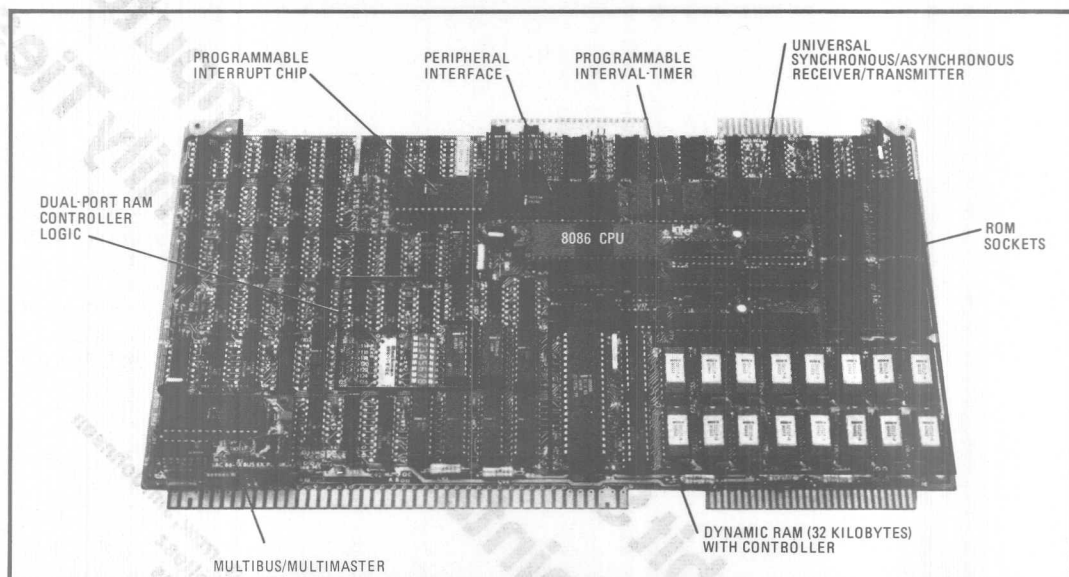
□ For the first time ever, 8- and 16-bit single-board computers can brainstorm over the same system bus. The iSBC 86/12 16-bit SBC has been designed to work intimately with its predecessors, the iSBC 80 family of 8-bit boards. What's in it for the user? Design flexibility—8-bit designs can be enhanced to 16 bits, developed software can be transported and, beyond that, 8- and 16-bit devices can be mixed in multiprocessing configurations. Several features make these options possible: a 16-bit CPU and instruction set designed for 8-bit compatibility; greatly expanded memory resources; and an extension of the Multibus specifications.

At the heart of the iSBC 86/12 is a 16-bit, high-performance metal-oxide-semiconductor 8086 central processing unit that operates at 5 megahertz. Because the 8086 instruction set is a superset to that of both the 8080A and 8085A 8-bit processors, the CPU can execute the full set of 8080A/8085A-type 8-bit instructions plus

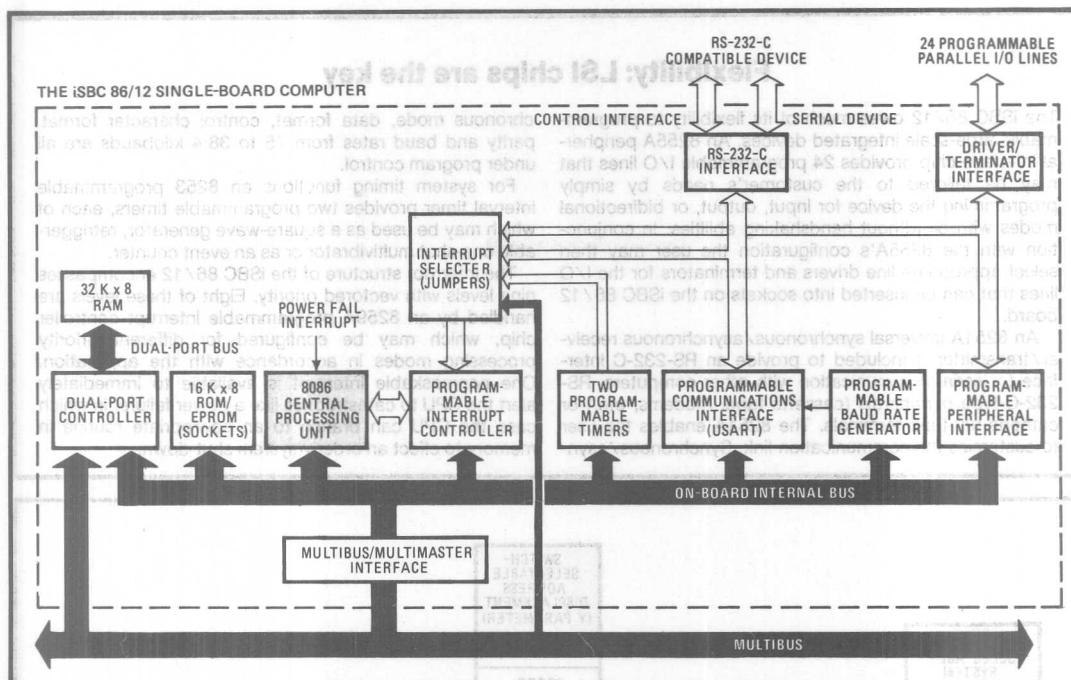
a new set of 16-bit instructions. Thus, programs generated for 8-bit-CPU systems can easily be upgraded to run on the iSBC 86/12 using the software tools available with the Intel microcomputer development system. Programs written in Intel's high-level programming language, PL/M, can be executed on both iSBC 80 and iSBC 86 products, preserving the software investment in 8-bit systems as a user moves into 16-bit applications.

Other features of the 8086 CPU are signed 8- and 16-bit arithmetic (including multiply and divide), efficient interruptible byte-string operations, and improved bit manipulation. Furthermore, the 8086 provides mechanisms for reentrant code, position-independent code, and dynamically relocatable programs.

This enhanced processing power is supported by the largest memory ever offered on a CPU board (Fig. 1). Memory address space has been extended over the iSBC 80 series to one million bytes. Up to 16 kilobytes of



1. What a board. The iSBC 86/12 has 32 kilobytes of RAM and room for 16 kilobytes of ROM. The 5-MHz 8086 CPU executes 8080A/8085A-type as well as 16-bit instructions, including multiply and divide. Address space has been increased to a megabyte.



2. LSI + SBC = 86/12. A number of programmable LSI devices take credit for the power and flexibility of the iSBC 86/12. Note their interconnection to the three-bus hierarchy. When the 8086 requests a resource, the system bus is used only as a last resort.

read-only memory can be installed on the iSBC 86/12 itself. Furthermore, an additional 32 kilobytes of dynamic random-access memory with on-board refresh may be accessed independently by the CPU or by the system bus (Multibus).

Like the iSBC 80/30, the 86/12's RAM has dual ports to extend its use off board for access by other Multibus masters, including single-board computers, direct-memory-access devices, and peripheral controllers [*Electronics*, Aug. 17, p. 109]. All memory operations on the board occur independently of the Multibus, freeing it for external parallel operations. For applications that require data integrity at all times, a separate bus supplies power to the RAM and support logic via the edge connector. An auxiliary power source energizes the RAM in the event of power failure.

Multibus—the new look

To exploit the greater performance of the 8086 CPU and simultaneously make the iSBC 86/12 fully compatible with the iSBC 80 family of SBCs and expansion products, the Multibus specification has been extended to support 20 bits of address and 16 bits of data. The control lines, too, have been expanded to direct 8- and 16-bit data transfer over the system bus. These improvements enable the iSBC 86/12 to address directly a full one megabyte of system memory, access data in 8- or 16-bit word lengths, and recognize and acknowledge a variety of interrupts.

Address space has been enlarged to 1 megabyte by adding four address lines, A_{10} – A_{13} . Next, 8- and 16-bit

data operations have been defined to permit both types in the same system. This is done by reorganizing the memory modules, adding one new signal and redefining another. The memory is divided into two 8-bit data banks, which form a single 16-bit word. The banks are organized such that all even-byte-addressed data is in one bank (D_0 – D_7) and all odd-byte-addressed data is in the other bank (D_8 – D_{15}). A new bus-address signal has been defined to control the odd-byte bank called byte high enable (BHEN) during 16-bit operations. When active, BHEN enables the high byte of the data word from the addressed boards on the D_8 – D_{15} Multibus data lines. A_0 controls the even byte bank and, when inactive, enables the low byte of the data word on the D_0 – D_7 Multibus data lines. All word operations must occur on an even-byte-address boundary with BHEN active for maximum efficiency. (A_0 is inactive for all even addresses—see the table.) Word operations on odd-byte boundaries will be converted to 2-byte operations by the 8086, one for low-byte, one for high-byte. Byte operations can occur in one of two ways. The even bank is accessed when BHEN and A_0 are both low. This puts the data on D_0 – D_7 . To access the odd bank (normally placed on D_8 – D_{15} during a word operation), a new data path has been defined. The active state of A_0 and the inactive state of BHEN are used to enable a swap-byte buffer, which places the odd data bank on D_0 – D_7 . This permits an 8-bit master access to both bytes of the data word while controlling only A_0 . A_0 therefore specifies a unique byte and is not part of the word address, since all word operations are on even-byte boundaries.

Flexibility: LSI chips are the key

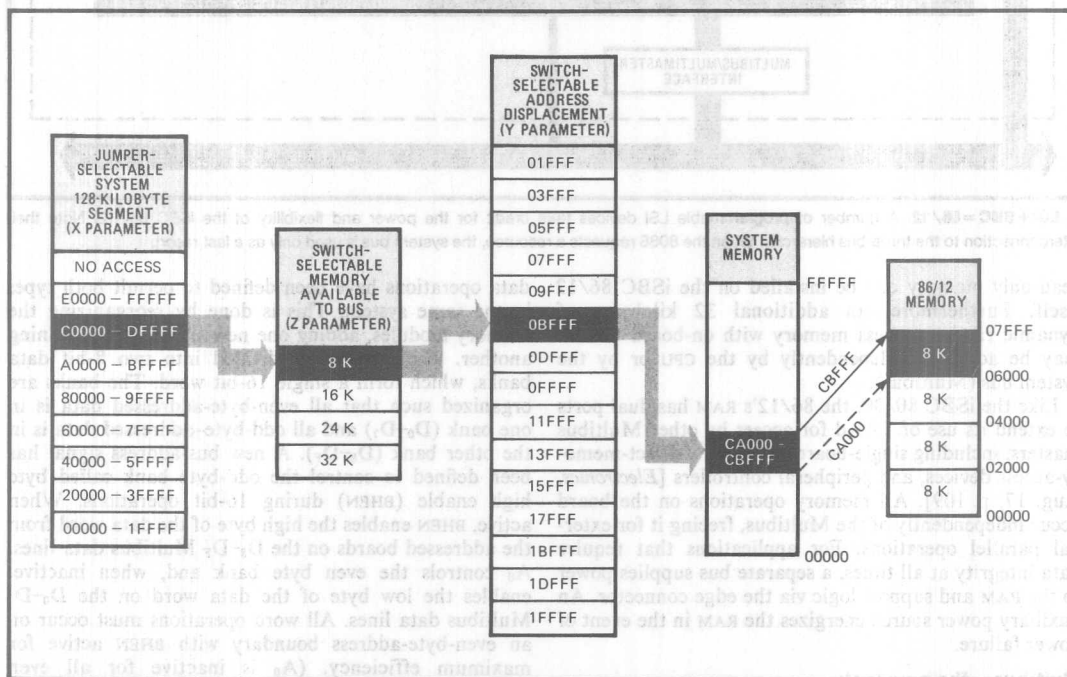
The iSBC 86/12 owes much of its flexibility to programmable large-scale integrated devices. An 8255A peripheral interface chip provides 24 programmable I/O lines that may be tailored to the customer's needs by simply programming the device for input, output, or bidirectional modes with or without handshaking abilities. In conjunction with the 8255A's configuration the user may then select appropriate line drivers and terminators for the I/O lines that can be inserted into sockets on the iSBC 86/12 board.

An 8251A universal synchronous/asynchronous receiver/transmitter is included to provide an RS-232-C interface for serial communication with other computers, RS-232-C-type peripherals (cassette tape, modems, etc.) or cathode-ray-tube terminals. The 8251A enables the user to customize the communication link. Synchronous/asyn-

chronous mode, data format, control character format, parity and baud rates from 75 to 38.4 kilobauds are all under program control.

For system timing functions an 8253 programmable interval timer provides two programmable timers, each of which may be used as a square-wave generator, retriggerable one-shot multivibrator or as an event counter.

The interrupt structure of the iSBC 86/12 encompasses nine levels with vectored priority. Eight of these levels are handled by an 8259A programmable interrupt controller chip, which may be configured for different priority processing modes in accordance with the application. One nonmaskable interrupt is available to immediately alert the CPU to catastrophes like a power failure, in which case the CPU can branch to an appropriate routine in memory to effect an orderly system shut-down.



3. RAM, please. The 8086's view of on-board memory is fixed from zero to 07FFFH. When an outside master accesses this space, the DP controller performs the translation. Here, locations 06000H to 07FFFH are available to another master by addressing CA000H to CBFFFH.

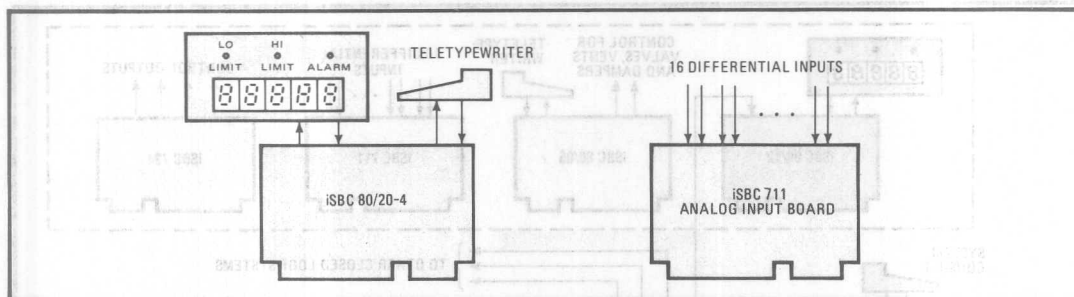
Since all 8-bit accesses via Multibus are done on the lower byte of the data word, the iSBC 86/12 can access 8-bit memory or I/O devices from the system bus. This makes the iSBC 86/12 compatible with all iSBC 80 Multibus modules.

More interrupts, too

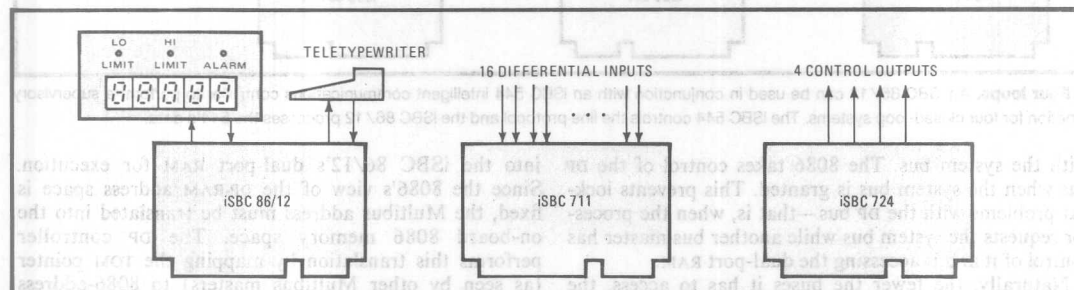
The iSBC 86/12 expands the previous Multibus definition of interrupts by creating two distinct types: non-bus-vectored (NBVI) and bus-vectored (BVI) interrupts. Each Multibus interrupt line can be individually defined

through software to be a BVI or NBVI. Using BVIs, the interrupt capability of a Multibus system can be increased to 64 bus-vectored-priority interrupts.

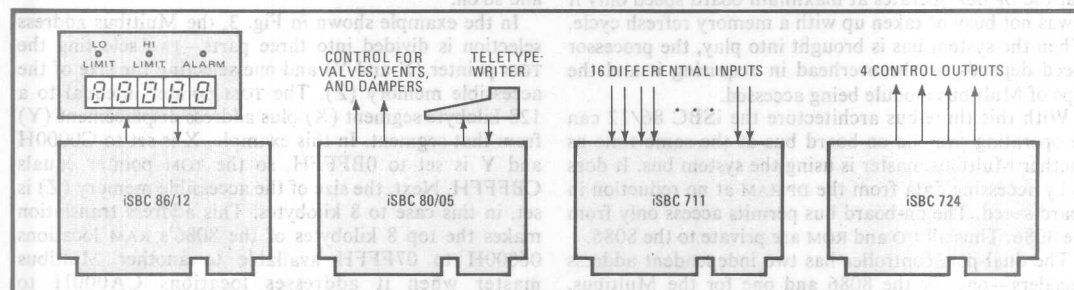
Using NBVIs, a slave module activates an interrupt line and the interrupted bus master generates its own restart address to service that interrupt. The Multibus address or data lines are not used. A BVI uses the Multibus address and data lines to communicate with the interrupting slave. When the slave module generates an interrupt, the bus master requires that module to generate the restart address. One additional command signal is



4. Open loop. Shown above is a simple alarm and monitoring system. The iSBC 711 analog-input board samples 16 differential inputs and the 8-bit iSBC 80/20 compares the inputs to the high and low limits. An alarm condition illuminates an LED and gets logged on a teletypewriter.



5. Closed loop. Suppose the system in Fig. 4 needs to be upgraded to handle a closed-loop system. For this application an iSBC 86/12 replaces the 80/20-04 to cope with the higher processing. The output control variables are handled by an iSBC 724 analog-output board.



6. Multi-master. To enhance the control system in Fig. 5, add a dedicated CPU to control valves, vents, and dampers that, in turn, affect pressure and flow parameters in the system. This has been done by adding an iSBC 80/05 in a Multibus/multimaster arrangement.

defined—interrupt acknowledge (INTA)—to request the restart address from the slave module.

The iSBC 86/12 board architecture, like that of the 8-bit iSBC 80/30, is organized around a three-bus hierarchy: an on-board bus, a dual-port bus and a system bus (Multibus). All three buses have been expanded over their 80/30 counterparts to incorporate 20 address lines and 16 data lines.

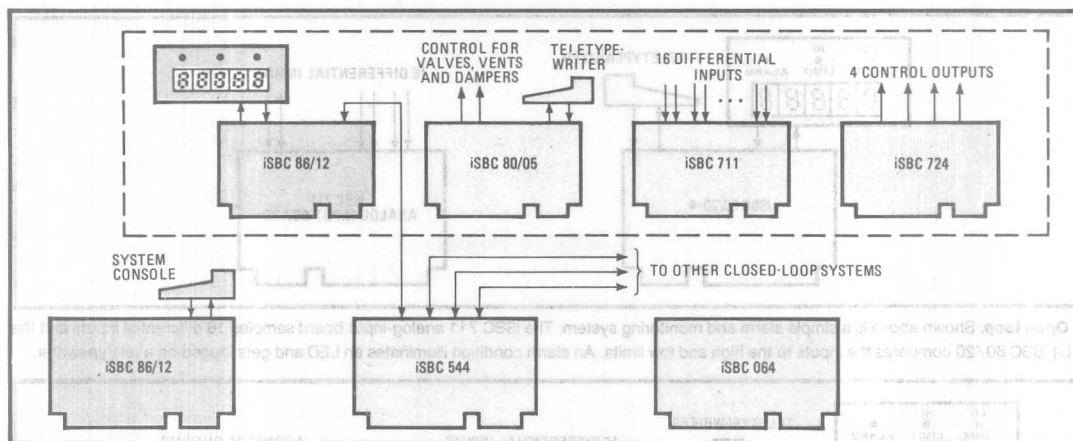
The iSBC 86/12 architecture

The on-board bus links the 8086, all the I/O peripherals, and the read-only memory. Next in the hierarchy is the dual-port bus, which connects to the DP controller, 32 kilobytes of dynamic RAM, and the dynamic RAM controller. Finally, the system bus permits expansion of system resources through Multibus modules (Fig. 2).

The bus protocol of the iSBC 86/12 dictates that each

of the three buses communicate with an adjacent bus or operate independently. When the CPU makes a request for a resource, the on-board and dual-port buses simultaneously determine if their hardware can fulfill the request. If the on-board bus is able to acknowledge the request, it does so and the DP bus is not disturbed. (The DP bus is not interrupted to determine whether it can acknowledge the request.) The 8086 always controls the on-board bus, and requested operations can be completed without delay. If the DP bus is needed, it is requested and the dual-port controller grants the use of the bus to the processor. Thereafter, the dynamic-RAM controller completes the operation and generates an acknowledge.

If neither the on-board nor the DP bus can satisfy the request, the CPU asks for the system bus. The 8086 must use the on-board and dual-port buses to communicate



7. Four loops. An iSBC 86/12 can be used in conjunction with an iSBC 544 intelligent communications controller to perform a supervisory function for four closed-loop systems. The iSBC 544 controls the line protocol and the iSBC 86/12 processes the 544's data.

with the system bus. The 8086 takes control of the DP bus when the system bus is granted. This prevents lock-out problems with the DP bus—that is, when the processor requests the system bus while another bus master has control of it and is accessing the dual-port RAM.

Naturally, the fewer the buses it has to access, the faster the iSBC 86/12 completes a transaction. The on-board bus always operates at maximum board speed. But the DP bus operates at maximum board speed only if it was not busy or taken up with a memory refresh cycle. When the system bus is brought into play, the processor speed depends on the overhead in acquiring it and the type of Multibus module being accessed.

With this three-bus architecture the iSBC 86/12 can be operating over its on-board bus at the same time as another Multibus master is using the system bus. It does so by accessing data from the DP RAM at no reduction in board speed. The on-board bus permits access only from the 8086. Thus all I/O and ROM are private to the 8086.

The dual-port controller has two independent address decoders—one for the 8086 and one for the Multibus. The 8086 decoder fixes the 8086's RAM addresses from hexadecimal 00000 to 07FFF using a fusible-link programmable ROM. The Multibus decoder allows the user to select any address range for the on-board RAM by specifying two parameters—a top-of-memory pointer and the size of the accessible memory. The TOM pointer (as seen by another Multibus master) can be set to any 8-kilobyte boundary in the 1-megabyte memory space. The amount of memory on the iSBC 86/12 accessible by another master can be set to 8, 16, 24, or 32 kilobytes (or no access) with an on-board jumper. For example, fixing the accessible memory size to 24 kilobytes provides the 8086 with 8 kilobytes of RAM that only it can access. This private area can be used for the processor's stack, interrupt jump table and other special system parameters that are generally protected from other Multibus masters. The only addressing restriction is that the memory block accessible to the Multibus cannot cross a 128-kilobyte boundary.

Suppose a Multibus master wants to load a program

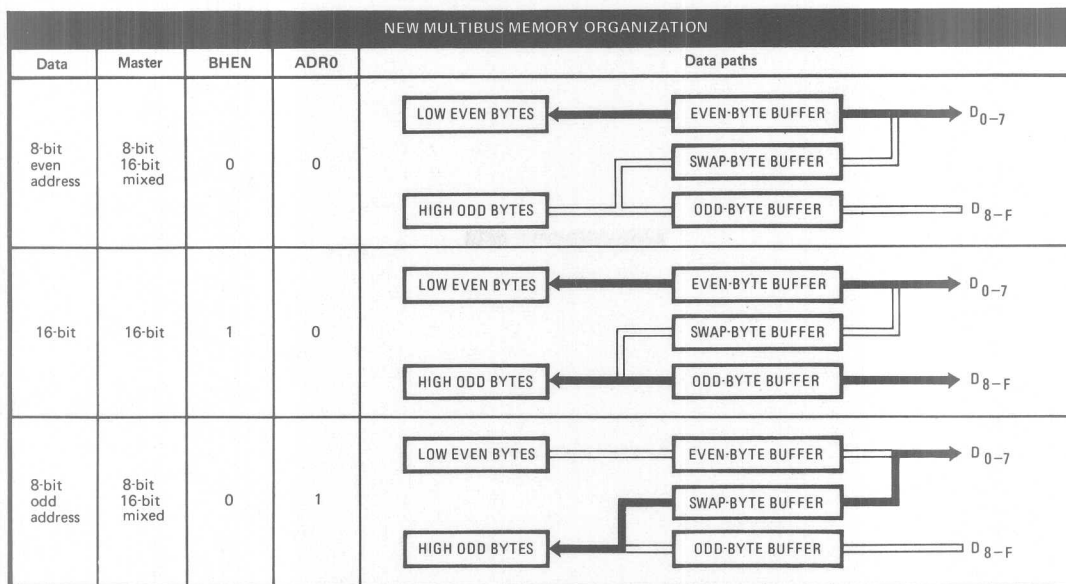
into the iSBC 86/12's dual-port RAM for execution. Since the 8086's view of the DP-RAM address space is fixed, the Multibus address must be translated into the on-board 8086 memory space. The DP controller performs this translation by mapping the TOM pointer (as seen by other Multibus masters) to 8086-address 07FFFH, the top of the 8086's on-board RAM. Pointer-1 is mapped to the top of 8086 on-board RAM-1, and so on.

In the example shown in Fig. 3, the Multibus address selection is divided into three parts—two selecting the TOM pointer (X and Y) and one selecting the size of the accessible memory (Z). The TOM pointer is equal to a 128-kilobyte segment (X) plus address displacement (Y) from that segment. In this example, X is set to C0000H and Y is set to 0BFFFH, so the TOM pointer equals CBFFFH. Next, the size of the accessible memory (Z) is set, in this case to 8 kilobytes. This address translation makes the top 8 kilobytes of the 8086's RAM locations 06000H to 07FFFH available to another Multibus master when it addresses locations CA000H to CBFFFH. The 8086 still has 24 kilobytes (00000H to 05FFFH) of private memory.

Multiprocessing schemes

In multiprocessing systems, a master must be able to access the system without another master obtaining the bus. The iSBC 86/12 incorporates bus-arbitration logic to effect these transactions. Since the system bus is only requested when a system resource is needed, the iSBC 86/12 can perform true parallel processing with other iSBC 80 or 86 masters.

A typical example is the use of a common memory location that contains the status byte (busy/not busy) of a floppy-disk controller. When the floppy disk is needed, the master must first read the location and, if not busy, write the status word without another master obtaining the bus (to use the floppy disk). A bus-lock function on the iSBC 86, once enabled, allows the iSBC 86 to maintain control of the system bus until the lock is disabled by program control. This bus-lock function may



be activated in one of two ways—by an output bit from the resident 8255A peripheral-interface chip or by a software prefix on any 8086 instruction. The iSBC 86 can perform the test and set function by exchanging the accumulator with the memory location, preceding the instruction by a lock prefix. For example, the status word is read into the accumulator and, without another intervening bus cycle, a busy status is written. The accumulator is then tested: if busy—try again (writing a busy does not destroy status as it was already busy); if not busy, the floppy disk is now under the master's control and the status location is set to busy.

The iSBC 86/12: a design tool

For system debugging and full-speed execution, the iSBC 86/12 can be linked to the Inteltec microcomputer development system. Programs generated on the Inteltec system can be downloaded into the iSBC 86/12 RAM via cables. Through a virtual-terminal capability, the Inteltec console can directly access an iSBC-resident monitor, which provides commands for software debug. Once the debugging cycle is completed, the user has the option of uploading the software back to the Inteltec for storage on diskettes.

The Multibus and form-factor compatibility of the 8-bit iSBC 80 and 16-bit iSBC 86 single-board computers provide a degree of design flexibility previously unobtainable. Initial design problems can be solved with low-cost 8-bit hardware. As product requirements evolve, 16-bit performance can be added. Eventually, 8- and 16-bit multiprocessor solutions can be conveniently implemented.

Consider the application shown in Fig. 4, an alarm and monitoring system in a typical process plant. Sixteen differential inputs from pressure and flow transducers are sampled once every second, then compared to high and low limits previously entered through thumbwheel

switches. The iSBC 711 analog-input board takes care of sampling the inputs and the 8-bit iSBC 80/20-4 compares the data to the high and low limits. Whenever these limits are exceeded, an alarm LED lights up and the alarm condition is logged on the system teletypewriter along with input identification, high limit, low limit and sampled value.

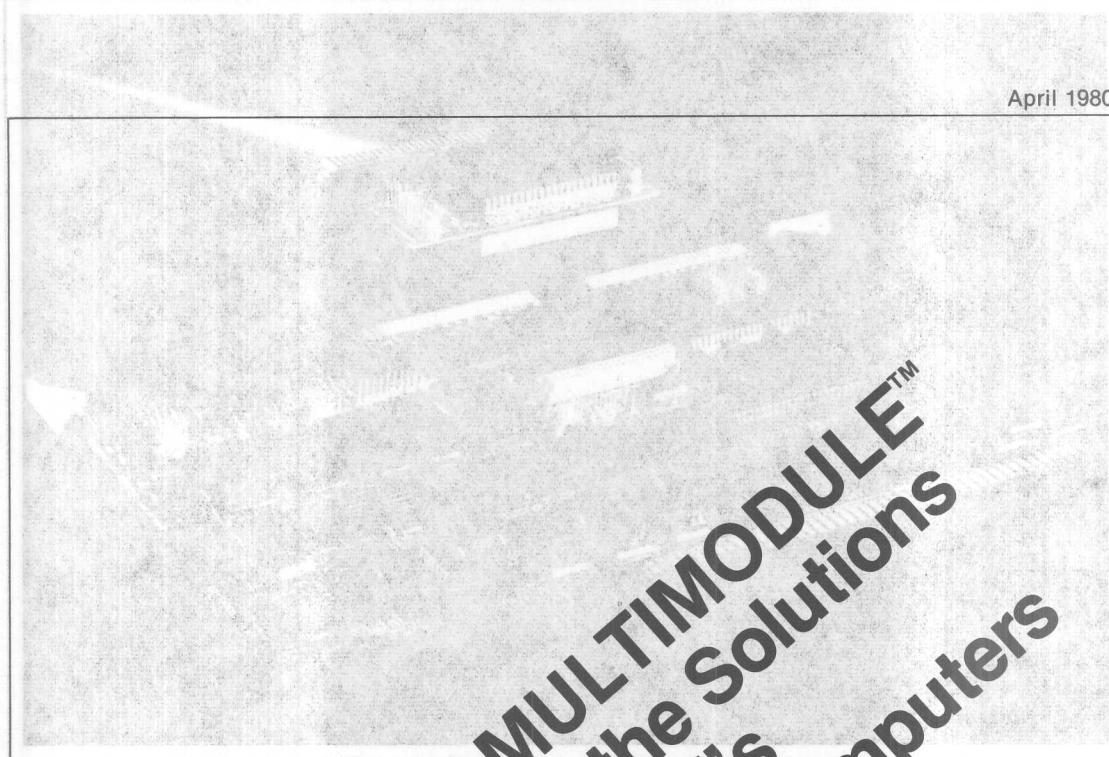
Closed loops

Instead of an open-loop system, suppose the design must be enhanced to control four output variables—thereby making it a closed-loop system. The sampling rate must be increased to once every third of a second and more processing will be required to run through the control algorithm and output the control-loop data. For this application, an iSBC 86/12 replaces the 80/20-4 to handle the higher processing requirements. An iSBC 724 analog-output board is also added to provide the four output-control variables (see Fig. 5). Carrying this example one step further, one may want to dedicate another processor to controlling valves, vents, and dampers that in turn affect pressure and flow parameters in the system. This can be done by adding an iSBC 80/05 in a multimaster arrangement as shown in Fig. 6.

Finally, an iSBC 86/12 can be used with an iSBC 544 intelligent communication-controller to supervise four closed-loop systems of the type shown in Fig. 6. The 86/12 of each system interfaces with the supervisory system via its serial interfaces, which are connected to the iSBC 544's serial ports (see Fig. 7). The iSBC 544 performs the control functions associated with the line protocol. The supervisory iSBC 86/12 can access the iSBC 544's dual-port memory and can perform further processing of the data received from the four closed-loop systems. In this configuration large amounts of memory may be required; since the iSBC 86/12 can address up to 1 megabyte, this presents no problem. □

A new family of MULTIMODULE™ boards extends the solutions provided by Intel's single board computers

April 1980



A New Family of MULTIMODULE™ Boards Extends the Solutions Provided by Intel's Single Board Computers

Preview Magazine, March/April 1980

FEATURE

A new family of MULTIMODULE™ boards extends the solutions provided by Intel's single board computers

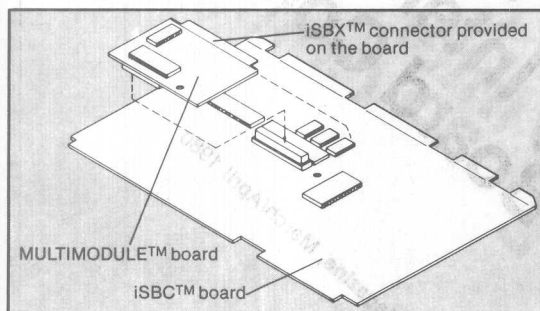
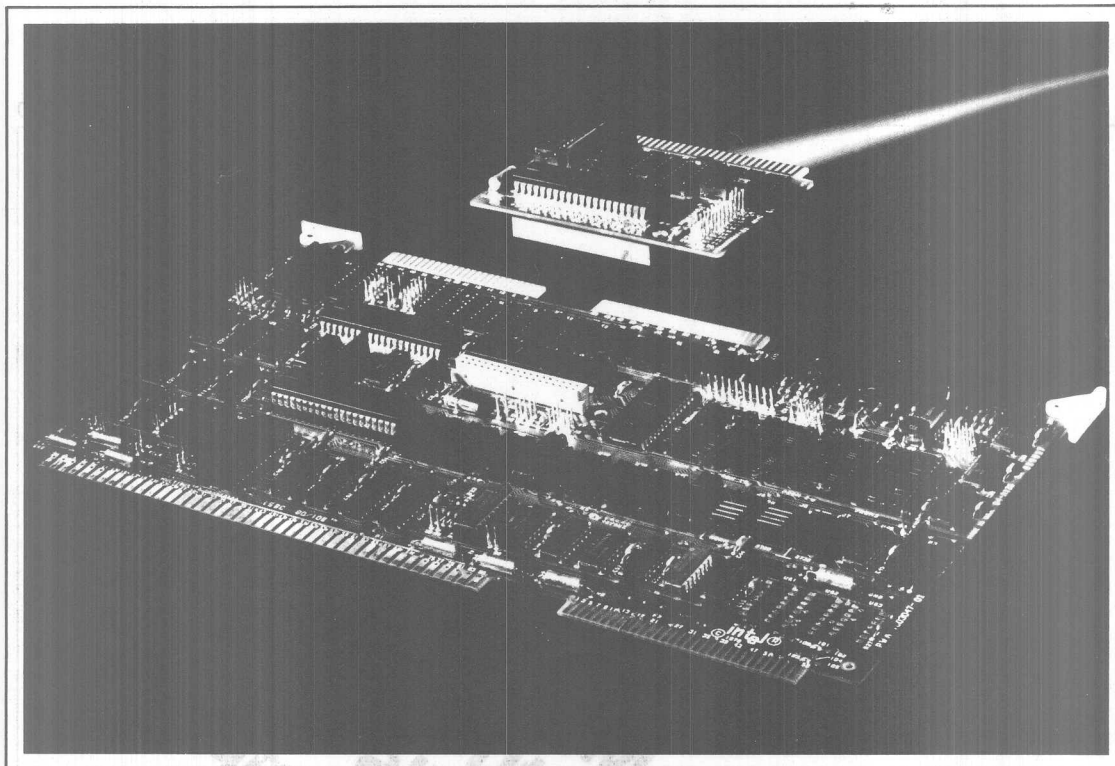


Figure 1. The iSBX 350™ Programmable I/O MULTIMODULE plug-in, shown here, connects directly on one end into the special iSBX bus connector and screws onto the edge of the single board computer on the other end.

A complete new family of products, called MULTIMODULE boards, has been announced by Intel Corporation. The new MULTIMODULE boards designed to extend the functional capabilities of single board computers at much lower cost than has been previously possible. Intel is supporting the MULTIMODULE concept with a new bus standard—the iSBX bus. The iSBX bus will now be designed into a new generation of single board computers to achieve compatibility with the emerging iSBX bus compatible MULTIMODULE product line. Users of Intel's single board computers can incrementally expand system resources by adding small (2.85" x 3.7") iSBX MULTIMODULE boards which plug directly into iSBC boards.

Three new iSBX bus MULTIMODULE plug-ins have been introduced—modules for parallel I/O, serial I/O,

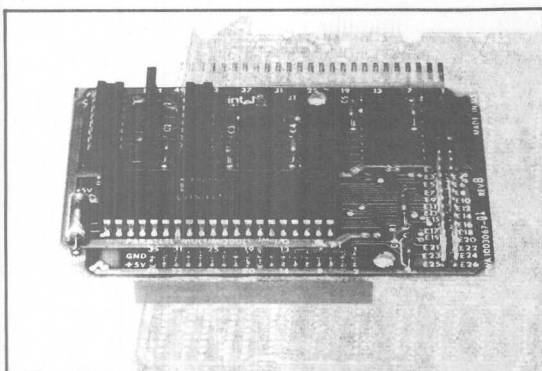


Figure 2. The iSBX 350™ Programmable I/O MULTIMODULE™ board provides 24 programmable I/O lines using the 8255A-5 programmable peripheral interface. Sockets are provided for interchangeable line drivers/terminators.

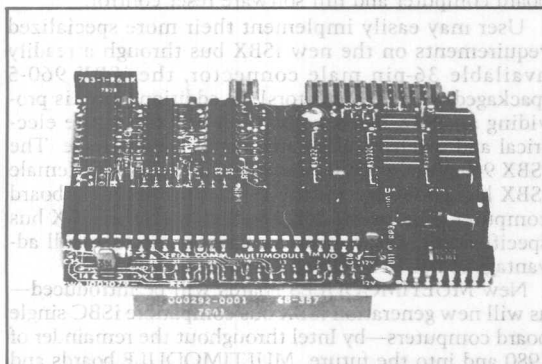


Figure 3. The iSBX 351™ Serial I/O MULTIMODULE board extends the serial communications capability of a single board computer via an Intel® 8251A USART. It includes an on-board programmable baud rate generator, two programmable 16-bit BCD/binary timers, and RS232C or RS422/449 interfacing.

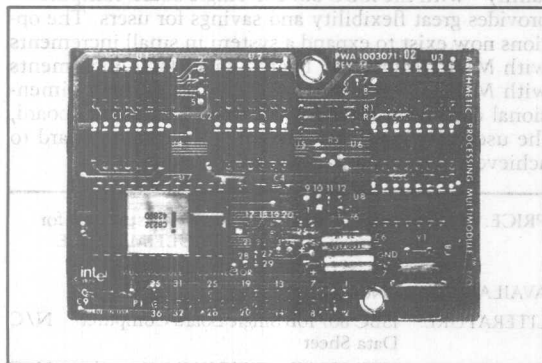


Figure 4. The iSBX 332™ Floating Point Math MULTIMODULE board uses an Intel® 8232 Floating Point Processor and is compatible with the new proposed IEEE format. It provides users with both single-precision (32-bit) and double-precision (64-bit) arithmetic functions.

and floating-point math. The showcase for the MULTIMODULE family is the iSBC 80/10B board—the first iSBX bus compatible single board computer.

MULTIMODULE™ boards allow users to tailor board level products

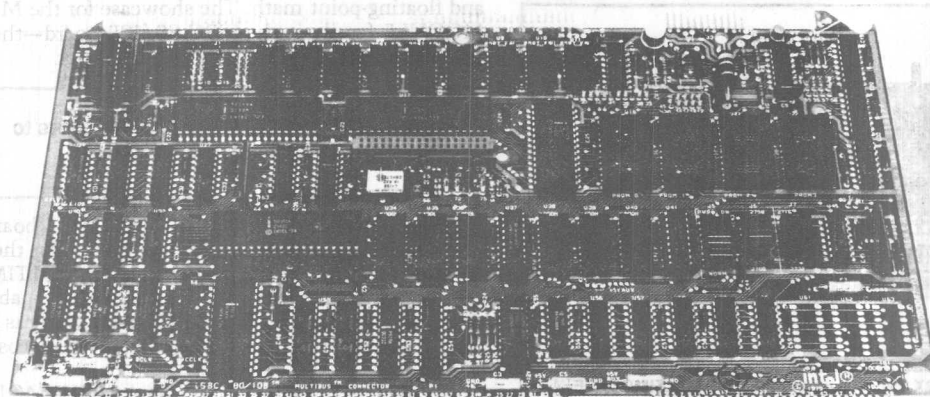
Customers can choose MULTIMODULE boards to precisely configure single board computers for their individual applications at a lower cost. The MULTIMODULE boards enable users to buy exactly the capabilities that they require for their iSBC-based systems. This can, of course, keep system size and system cost at a minimum.

Any of the three new iSBX bus MULTIMODULE products—the iSBX 350 Programmable I/O, the iSBX 351 Serial I/O, and the iSBX 332 Floating Point Math board—plug into a special interface called the iSBX bus on the iSBC 80/10B single board computer, (see Figure 1). The iSBX bus is being introduced by Intel to facilitate the interface between the iSBX MULTIMODULE boards and the iSBC products. The iSBX bus will become a standard, similar to the standard MULTIBUS system architecture. The new iSBX bus allows system expansion through the new MULTIMODULE boards without making any demands on the system's MULTIBUS interface. As a result, the system design achieves maximum on-board performance while freeing up the MULTIBUS interface for other system activities.

iSBC 80/10B™ board is customer-expandable

The new iSBC 80/10B single board computer is fully compatible with the popular iSBC 80/10A, but it is customer-expandable in "all directions." This board can be expanded in three dimensions to tailor EPROM, RAM, and I/O needs directly on the board. This gives a user a built-in adaptability on one board that was previously unattainable. First, the user can choose four types of Intel PROMs—the 2708, 2758, 2716 or 2732—expandable up to 16K bytes. Second, 1K of RAM memory is included on-board and may now be extended up to 4K with the use of standard Intel 2114A-5 1K x 4 static RAMs. Sockets on the 80/10B board are provided for these additional RAMs. Input/output is the third dimension in every single board computer and may now be expanded on-board the iSBC 80/10B by using one of the iSBX bus MULTIMODULE boards. In addition, a 1.04 millisecond timer has been added as a standard feature of the iSBC 80/10B board. In comparison, the iSBC 80/10A board has only 1K of on-board RAM, 8K of EPROM memory, no I/O or memory expansion facilities and no timer.

Two of the new MULTIMODULE boards—the iSBX 350 and iSBX 351—provide expansion to the I/O already on the iSBC 80/10B board. The iSBX 350 Programmable I/O MULTIMODULE (see Figure 2) provides 24 I/O lines with sockets for interchangeable line



iSBC 80/10B™ Single Board Computer

Here's what iSBX™ bus MULTIMODULES™ provide the user:

- The ability to incrementally expand the iSBC Single Board Computer, via iSBX bus, allowing the user to add only the function his application requires.
- The on-board addition of totally new capabilities (mathematics processing and the like) to single board computers.
- Maximum performance by reducing traffic required for standard expansion boards on the industry standard MULTIBUS interface.
- Compatibility with future 8-bit and 16-bit single board computers from Intel
- A high-reliability, 36-pin iSBX 960-5 male connector for customer design

drivers and terminators. The iSBX 351 Serial I/O MULTIMODULE (see Figure 3) provides programmable, synchronous/asynchronous RS232C or RS422/449 compatible serial expansion with fully software selectable baud rate generation. Additionally, two programmable 16-bit BCD and binary timers are available, allowing iSBC 80/10B users to implement programmable timing functions directly on the board.

A third MULTIMODULE board which users may select, provides additional on-board mathematics capability to their single board computer. The iSBX 332 Floating Point Math MULTIMODULE board (see Figure

New iSBX 960-5™ MULTIMODULE™ connectors are available off-the-shelf

4) is compatible with the proposed IEEE format standard. It offers single-precision (32-bit) and double-precision (64-bit) arithmetic functions including Add, Subtract, Multiply and Divide. It also includes end-of-operation and error interrupts to its host iSBC single

board computer and full software reset control.

User may easily implement their more specialized requirements on the new iSBX bus through a readily available 36-pin male connector, the iSBX 960-5 (packaged with 5 connectors). In addition, Intel is providing an iSBX bus specification describing the electrical and mechanical parameters of the interface. The iSBX 960-5 male connectors mate directly to the female iSBX bus connector on the iSBC 80/10B single board computer. The connector, together with the iSBX bus specification, allows system designers to take full advantage of MULTIMODULE benefits.

New MULTIMODULE boards will be introduced—as will new generation iSBX bus compatible iSBC single board computers—by Intel throughout the remainder of 1980 and into the future. MULTIMODULE boards and the iSBX bus represent a major commitment by Intel for the future and offer system designers new options to complement single board computers and MULTIBUS expansion.

The immediately available MULTIMODULE family—with the iSBC 80/10B single board computer—provides great flexibility and savings for users. The options now exist to expand a system in small increments with MULTIMODULE boards, or in large increments with MULTIBUS boards. And, with the three dimensional expansion capability of the iSBC 80/10B board, the user may configure his entire system on-board to achieve a single board, low cost solution.

PRICE: Call your local Intel sales office or distributor for the latest prices on all of the MULTIMODULE products

AVAILABILITY: Now

LITERATURE: iSBC 80/10B Single Board Computer N/C
Data Sheet

iSBX 350 MULTIMODULE Data Sheet N/C

iSBX 351 MULTIMODULE Data Sheet N/C

iSBX 332 MULTIMODULE Data Sheet N/C

Special-function modules ride on computer board

Smaller cards donate floating-point processing or added serial and parallel I/O to primary single-board computer; memory is extensible on the main card

by Gary Sawyer, Jim Johnson, David Jurasek, and Steve Kassel, Intel Corp., Hillsboro, Ore.

□ In the design of board-level computers, two basic methods coexist. One is to pack each card with integrated circuits to the limits of its capacity, and the other is to distribute the computer functions among other boards occupying additional card slots.

Both approaches have their advantages. The single powerful module conserves space and expensive connectors, while the decentralized boards allow the user to pick and choose functions—and add them incrementally—although the expense of one board might spell overkill for one particular application.

A new concept in single-board computer architecture strikes a neat compromise between both camps. Rather than cram more chips on an already overstuffed board, the idea is simply to provide it with a connector for plugging in smaller modules having limited functions for specialized applications.

Best of both worlds

This is the idea behind iSBX Multimodule boards, which cost from \$155 to \$450 apiece. Plugging into a primary processor card, 10.5-square-inch boards with various types of memory or input and output functions provide the larger single-board computer with more versatility. Linking the base board and these Multimodule boards is a new 36-line bus called the iSBX bus, for single-board expansion. This interface is destined to match the popularity of the main board's Multibus interface connector (Fig. 1).

The iSBX bus is derived directly from the on-board microprocessor system bus and, as such, an iSBX-compatible board becomes an integral element of the single-board computer. The physical interface uses a unique connector designed specifically for the iSBX bus. The bus is brought to a female connector on the single-board computer; its male equivalent is resident on the iSBX board (Fig. 2).

The iSBC 80/10B board in Figs. 1 and 2 is the first single-board computer to be compatible with the iSBX bus. Upwardly compatible with its predecessor, the iSBC 80/10A, the iSBC 80/10B is functionally equivalent but offers significant enhancements.

The iSBC 80/10B board offers direct functional expansion in three dimensions—not only read-only memory (as in the iSBC 80/10A), but also static random-access memory and input and output, as facilitated by

the Multimodule boards. One kilobyte of static RAM is provided along with sockets for expansion in increments of 1-K bytes to 4-K bytes using standard 2114A-5 memories. Read-only memory may be expanded with standard ultraviolet-light-erasable and mask-programmable types to 16-K bytes.

The iSBC 80/10B also features an on-board 1.04-millisecond timer with ongoing clocking that users may optionally configure for microprocessor interrupts. In addition, power-fail control is provided for the 2114A-5 static RAMs, enabling the user to add battery backup if the memory contents must be preserved.

Three Multimodule boards

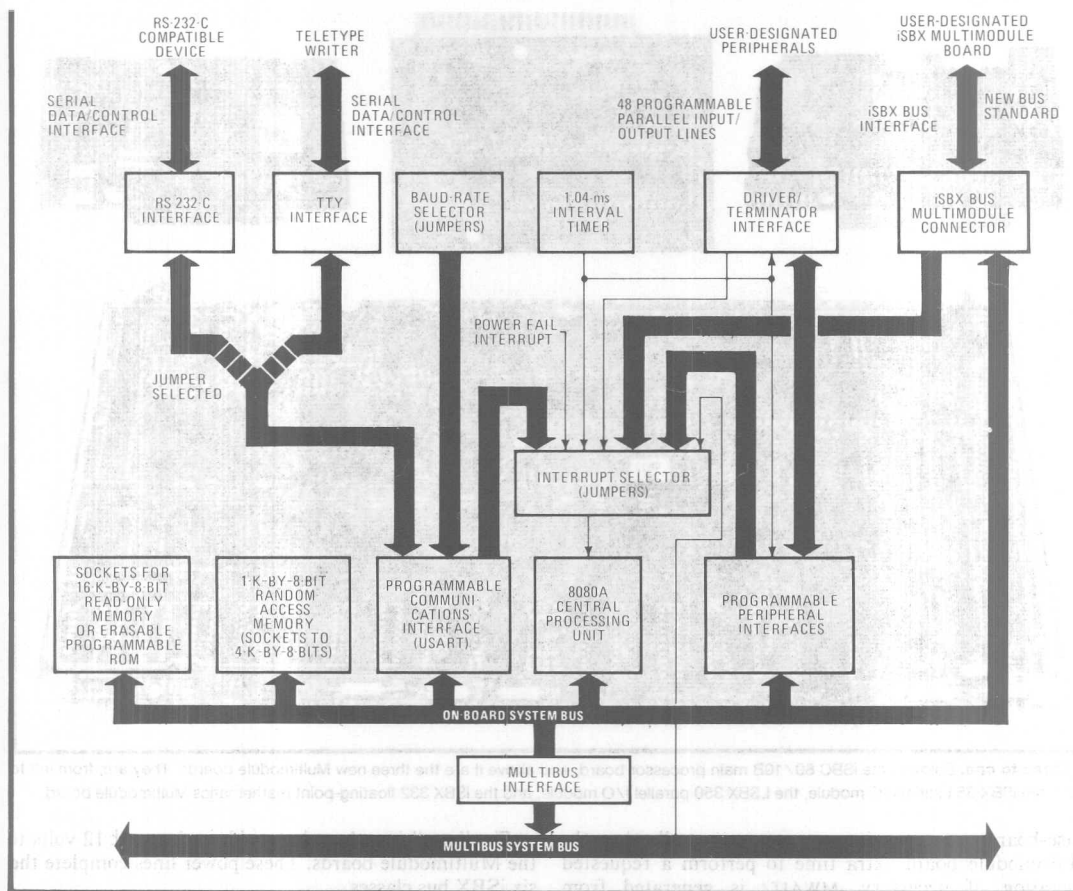
Being introduced along with the iSBC 80/10B are three Multimodule boards that expand the functional capacity of the single-board computer. Two of these, the \$155 iSBX 350 and \$230 iSBX 351, provide the same kind of input/output functions as are to be found on the processor board, only more of them.

For example, the 48 programmable I/O lines on the iSBC 80/10B board may be expanded to 72 lines by simply plugging in the iSBX 350 module—a 50% increase. Serial I/O is similarly expanded with the iSBX 351 module, which provides a programmable universal synchronous-asynchronous receiver/transmitter, or Usart (an 8251A), for compatibility with the RS-232-C and RS-449/422 interfaces. The iSBX 351 module further offers software-selectable baud rates and two programmable 16-bit binary or binary-coded decimal timers.

The third Multimodule board adds otherwise unavailable high-speed math capabilities to the iSBC 80/10B board. The \$450 iSBX 332 board uses the 8232 floating-point processor for arithmetic compatible with the standard currently being proposed by the Institute of Electrical and Electronics Engineers.

Many applications require a custom design. To complement the standard family of Multimodule boards, the iSBX 960-5 is provided. This includes five male iSBX connectors, and a full bus specification is available for custom interfacing by the user. This combination permits a user to satisfy his or her requirements for specialized I/O interfaces with the Multimodule concept.

The Multimodule concept can be divided into two logical elements: base boards and Multimodule boards.



1. Distributed. The first processor card to receive the iSBX bus connector is the iSBC 80/10B, a follow-on to the iSBC 80/10A. The off-board system bus is the Multibus, which interfaces to the on-board system bus. This, in turn, connects to the Multimodule board connector.

The base board is the master of the system in that it controls communication between the base's microprocessor and the Multimodule board's port. Though the first base board is a single-board computer, the iSBC 80/10B - Multibus-compatible slaves and intelligent I/O boards will also incorporate iSBX bus interfaces. The Multimodule board is a slave of the system in that it carries out I/O commands from the base board.

The iSBX interface

The iSBX bus specification includes both electrical and mechanical characteristics. The mechanical interface is convenient and rugged; the Multimodule board is mounted to the base board in two places, at the top with a screw and at the bottom by the iSBX bus connector. The connector is extremely reliable. It has gold-plated phosphor-bronze contacts, it is keyed to assure proper orientation, and a shroud protects its pins during handling. The connector also incorporates interlocking tabs to ensure a solid mechanical interface.

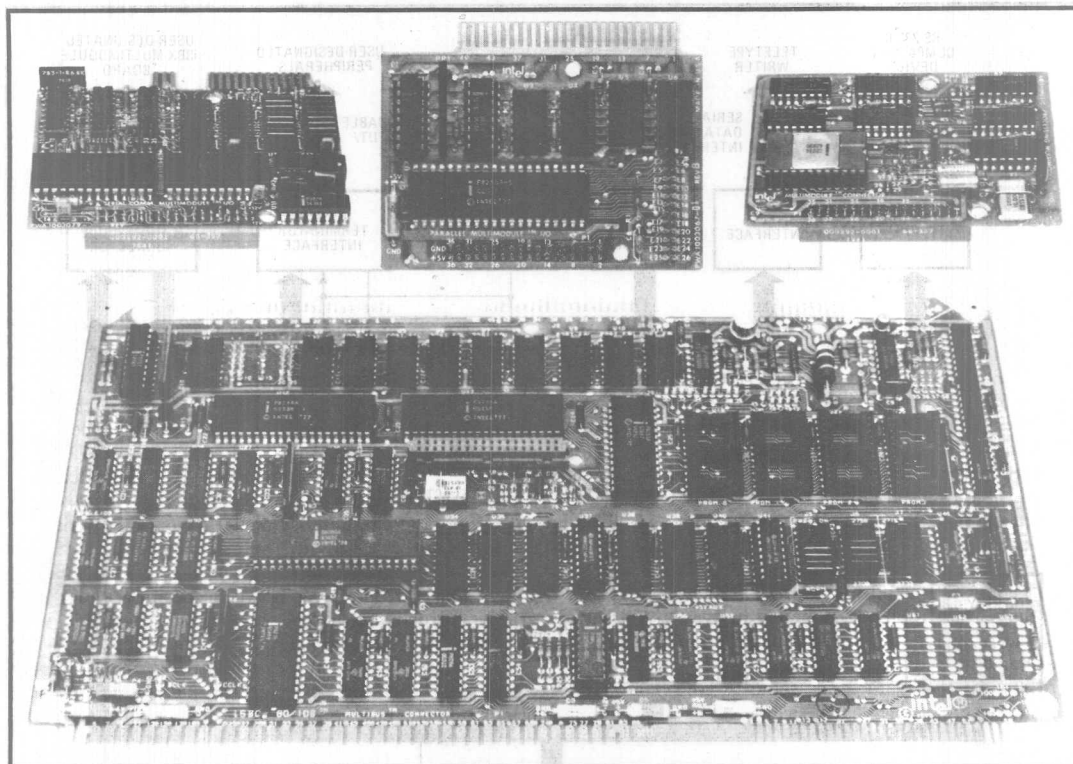
Electrically, the iSBX bus interface lines can be

grouped into six classes—control, address and chip select, data, interrupts, options, and power—for a total of 36 signal lines.

Control lines can be further grouped into those for commands, initialization, a clock, and system control. The two command lines (IORD/ and IOWRT/) are active-low I/O-read and -write signals that control the communication link between the base board and the Multimodule board. With a chip-select signal, an active command line indicates that the address lines are valid and that the Multimodule board should perform a specified operation.

The initialize line (reset) is an active-high input line from the base board that puts the Multimodule board into a known internal state. The clock line (MCLK) has a frequency of 10 megahertz, +0% or -10%. Being asynchronous with respect to all other Multimodule signals, this frequency can vary from base board to base board.

The remaining control lines, MWAIT/ and MPST, are output signals from the Multimodule board that control the state of the system. MWAIT/, active low, puts the



2. Three to one. Below is the iSBC 80/10B main processor board, and above it are the three new Multimodule boards. They are, from left to right, the iSBX 351 serial I/O module, the LSBX 350 parallel I/O module, and the iSBX 332 floating-point mathematics Multimodule board.

base-board processor into a wait state, allowing the Multimodule board extra time to perform a requested operation, if necessary. M_{WAIT} is generated from address and chip-select information only. MPST is tied to ground on the Multimodule board to inform the base board that a Multimodule board has been installed.

The second class of iSBX bus lines includes the address lines (MA₀-MA₂) and the chip-select lines (MCS₀ and MCS₁). The base board decodes I/O addresses to generate the chip-select signals for the Multimodule boards. In so doing, it normally decodes all but the three lowest-order addresses (MA₀-MA₂). A base board normally reserves two blocks of eight I/O ports for each iSBX bus connector provided.

Defining the lines

Eight bidirectional data lines (MD₀-MD₇, active high) carry information to and from the Multimodule ports. MD₀ is the least significant bit. The two active high interrupt lines from the Multimodule board, MINTR₀ and MINTR₁, make interrupt requests to the base board.

Two optional lines, OPT₀ and OPT₁, are connected to wire-wrapped posts on both the base and Multimodule boards. They may serve either as additional interrupts from the Multimodule board or as special signals from the base board.

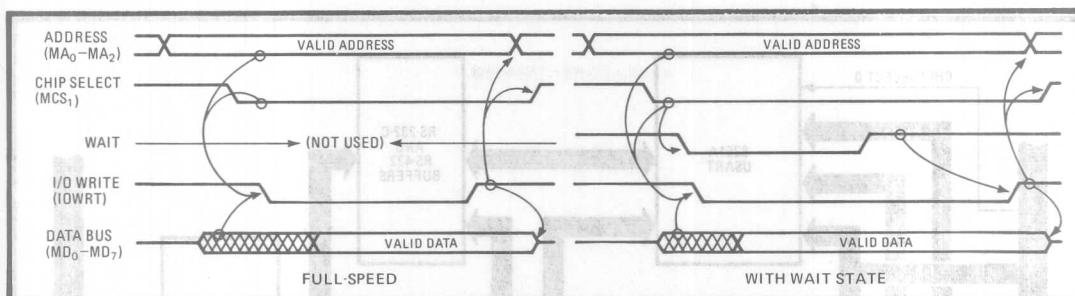
Finally, all base boards provide +5 and ± 12 volts to the Multimodule boards. These power lines complete the six iSBX bus classes.

The primary function of the iSBX bus is to provide a path for I/O-mapped data between base board and Multimodule board. This happens when the base board performs an I/O-read or I/O-write operation. There are two types of I/O-write operations, and the Multimodule board determines which is performed.

Data transfers

The first is a full-speed I/O write (Fig. 3). The base board generates a valid I/O address and chip-select and activates the IOWRT line after the set-up times are met. The IOWRT line will remain active for a minimum of 300 nanoseconds and the data will be valid for a minimum of 250 ns before IOWRT is removed. The base board then removes the data, address, and chip-select signals after the hold times shown in the timing diagram.

The alternative I/O write is a write-with-wait, used by Multimodule boards that cannot write into an I/O port at full speed. Again, the base board generates a valid address and chip-select. The Multimodule board activates the M_{WAIT} signal based on address and chip-select information. This will remove the ready condition from the processor, causing it to go into a wait state after the write command has been activated and valid data



3. Write types. Two modes of sending information to a Multimodule board are available: full-speed and with a wait state. The wait line is not used for peripherals that meet the full-speed specifications. The wait signal extends the time for which data remains valid.

provided. The Multimodule board will remove the MWAIT/ signal—allowing the processor to leave its wait state—when it has satisfied the write-pulse-width requirement. The base board removes the write command, then the data, address, and chip-select signals, after the hold times are met.

There are two types of I/O-read operations as well, and again they are determined by the Multimodule board. The first is a full-speed I/O read (Fig. 4). The base board generates a valid I/O address and chip-select, and after the set-up timings are met, it activates the IORD/ line. The Multimodule board must generate valid data from the addressed I/O port in less than 250 ns. The base board reads the data and removes the command, address, and chip-select signals as indicated in the timing diagram.

Read-with-wait, whose timing is at right in Fig. 4, is used by Multimodule boards that cannot perform a read operation under the full-speed specifications. The base board generates a valid address and chip-select, just as with a full-speed read. However, the Multimodule board now activates the MWAIT/ signal, which in turn removes the ready input to the base's processor, putting it into a wait state. The processor activates the IORD/ signal before going into a wait state.

The Multimodule board will remove the MWAIT/ signal when valid data can be read from the data bus. After reading the data, the base board removes the command, address, and chip-select signals.

The iSBX 351 serial I/O board is a good example of how easily large-scale integrated circuits may be interfaced by the iSBX bus. It presents the iSBC 80/10B

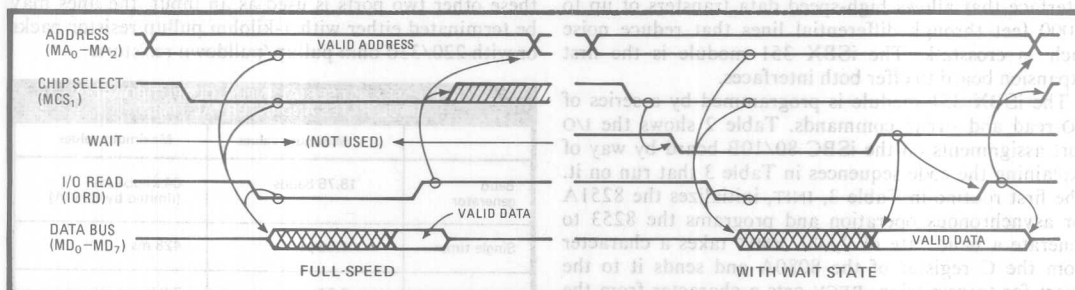
board with a second serial port.

The iSBC 351 board (Fig. 5) provides a synchronous or asynchronous serial communications channel with programmable format and baud rates up to 64 kilobits per second. In the synchronous mode, the user selects via software the number and format of the synchronization characters and the number of data bits. Parity may be even, odd, or disabled. In the asynchronous mode, the number of data bits and stop bits, as well as parity generation and detection, may be specified under program control. The added channel is compatible with either the RS-232-C or RS-422/449 interface.

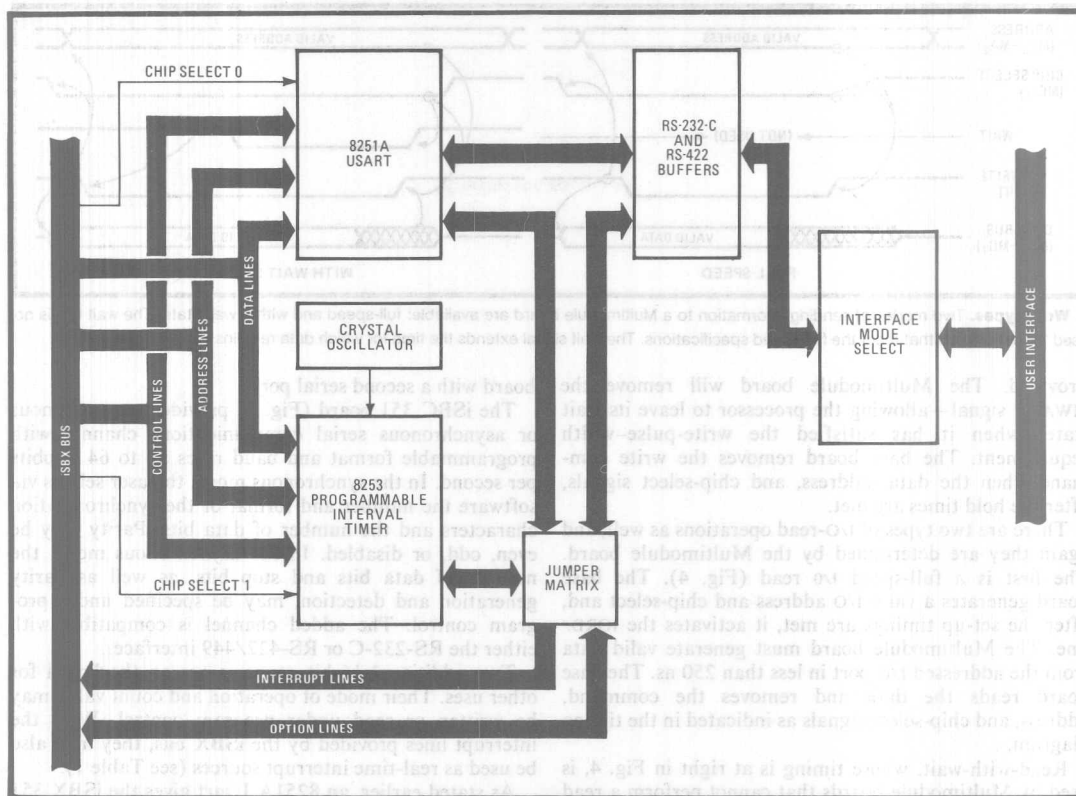
Two additional 16-bit counters are on the board for other uses. Their mode of operation and count value may be written or read under program control. With the interrupt lines provided by the iSBX bus, they may also be used as real-time interrupt sources (see Table 1).

As stated earlier, an 8251A Usart gives the iSBX 351 module a high-performance communications channel. In addition, an 8253 programmable interval timer (PIT) provides the three counters for clock generation and timing. Note in the block diagram of Fig. 5 that both devices are connected directly to the data, address, and command buses with no buffers. (Each chip, however, has its own chip-select line, preventing data bus contention.) The absence of buffers keeps the parts count down and the speed up.

Also shown in the extra block diagram are two optional lines that may be used as additional interrupt lines or to interface to the additional timer/counters. There are four interrupt sources on the board. Two, from the 8251A, indicate either that a character has been received



4. Read types. As in writing data into a Multimodule board, information is read from it in one of two ways: full speed or read-with-wait. The wait state extends the length of time that data is valid. This is necessary for slower transactions such as analog-to-digital conversions.



5. More serial I/O. The iSBX 351 serial input/output Multimodule board provides the base processor with an additional synchronous or asynchronous communications channel—one that is compatible with either the RS-232-C or the RS-422/449 interface specifications.

for reading or that the transmitter buffer is empty and ready for transmission. Two other interrupts may be generated by the timer/counters. The timers count from an on-board crystal-controlled oscillator.

Compatible with both

The iSBX 351 module uses a unique split-edge connector to provide compatibility with both RS-232-C and RS-449 interfaces. RS-232-C is commonly used to communicate with terminals, modems, and other equipment up to a distance of 50 feet away. RS-422 is a new interface that allows high-speed data transfers of up to 4,000 feet through differential lines that reduce noise such as crosstalk. The iSBX 351 module is the first expansion board to offer both interfaces.

The iSBX 351 module is programmed by a series of I/O-read and -write commands. Table 2 shows the I/O port assignments on the iSBC 80/10B board by way of explaining the code sequences in Table 3 that run on it. The first routine in Table 3, INIT, initializes the 8251A for asynchronous operation and programs the 8253 to generate a baud rate of 9,600. XMIT takes a character from the C register of the 8080A and sends it to the Usart for transmission. RECV gets a character from the Usart and places it in the accumulator. Note that in both data-transfer routines the Usart status register is checked

to ensure proper operation.

The iSBX 350 programmable I/O Multimodule board provides 24 general-purpose I/O lines (or three 8-bit ports) via a standard 50-pin edge connector, giving the iSBC 80/10B a total of 72 I/O lines. The 8255A is the only LSI component on the board, and six sockets are provided for line drivers or terminators.

Two bidirectional inverting 4-bit bus transceivers are provided for one of the three ports; sockets for the other two are TTL-compatible, allowing the use of inverting, noninverting, or open-collector drivers. When either of these other two ports is used as an input, the lines may be terminated either with 1-kilohm pullup resistor packs or with 220/330-ohm pullup/pulldown resistors.

TABLE 1: iSBX 351 SERIAL INPUT/OUTPUT BOARD'S BAUD RATES AND INTERVAL TIMES

	Minimum values	Maximum values
Baud generator	18.75 bauds	64 kilobauds (limited by 8251A)
Single timer	1.63 μ s	428 ms
Dual cascaded timers	3.26 μ s	7.8 h

TABLE 2: iSBX 351 ADDRESS ASSIGNMENTS

Universal synchronous/asynchronous receiver-transmitter	data port control port	F0, F2, F4, or F6 F1, F3, F5, or F7
Timer	counter 0 counter 1 counter 2 control port	F8 or FC F9 or FD FA or FE FB or FF

TABLE 3: SERIAL INPUT/OUTPUT ROUTINES

INT:	MVI	A, 96	Mode word to 8253 counter 2
OUT		0FB	
MVI	A, 8		Divide value to 8253 counter 2
OUT		0FA	
MVI	A, 0FE		Mode word to 8251A Usart
OUT		0F1	
MVI	A, 27		Command word to 8251A
OUT		0F1	
RET			
XMIT:	IN	0F1	Check Usart status to make sure it's ready to transmit a character
ANI		01	
JZ	XMIT		Loop until ready
MOV	A, C		Get data character
OUT		0F0	Send it
RET			
RCV:	IN	0F1	Check Usart status to see if a new character has been received
ANI		02	
JZ	RCV		Loop until data is available
IN		0F1	
ANI		38	Check framing, overrun, and parity error bits
JNZ	ERROR		Jump to an error handler if there are any problems
IN		0F0	Get data
RET			

The iSBX 350 module supports all three 8255A modes: basic I/O, strobed I/O, and strobed bidirectional bus I/O. Several of the handshaking signals are available as interrupt sources, and an additional external interrupt may be brought in via the edge connector.

Programming this board is as simple as programming the 8255As on the iSBC 80/10B itself. First, a mode word is written to the control port to specify the operational mode for each port. Data transfer may then begin, in the form of I/O-read or -write operations.

The iSBX 332 module is an accurate 32- or 64-bit floating-point processor that performs arithmetic operations in accordance with the proposed IEEE floating-point standard. It uses the 8232 floating-point processor.

The math module uses one data format that has two word lengths of 32 or 64 bits. The board will add, subtract, multiply, and divide for both word lengths.

TABLE 4: COMMAND MNEMONICS OF iSBC 332 FLOATING-POINT MATH MULTIMODULE

Command type	Mnemonic	Command description ¹
32-bit	SADD	Add TOS to NOS. Result to NOS. Pop stack.
32-bit	SSUB	Subtract TOS from NOS. Result to NOS. Pop stack.
32-bit	SMUL	Multiply NOS by TOS. Result to NOS. Pop stack.
32-bit	SDIV	Divide NOS by TOS. Result to NOS. Pop stack.
64-bit	DADD	Add TOS to NOS. Result to NOS. Pop stack.
64-bit	DSUB	Subtract TOS from NOS. Result to NOS. Pop stack.
64-bit	DMUL	Multiply NOS by TOS. Result to NOS. Pop stack.
64-bit	DDIV	Divide NOS by TOS. Result to NOS. Pop stack.
—	CLR	Clear status register.
32-bit	CHSS	Change sign of single-precision operand on TOS.
64-bit	CHSD	Change sign of double-precision operand on TOS.
32-bit	PTOS	Push single-precision operand on TOS to NOS.
64-bit	PTOD	Push double-precision operand on TOS to NOS.
32-bit	POPS	Pop single-precision operand from TOS. NOS becomes TOS.
64-bit	POPD	Pop double-precision operand from TOS. NOS becomes TOS.
32-bit	XCHS	Exchange single-precision operands TOS and NOS.

1. abbreviations: NOS = next on stack, TOS = top of stack

Table 4 shows the instruction mnemonics and functions, as well as the positions in the stack (top of stack or next on stack) the operands and results occupy.

The 8232 runs at 4 MHz for maximum throughput. A multiplication of two 32-bit quantities takes about 50 microseconds, excluding data entry and retrieval. In addition, two interrupts signal the base-board processor of completion of an operation or an error.

Floating-point math

The two word lengths of the floating-point standard were chosen for the highest speed and accuracy. If speed is the primary objective, the 32-bit format gives a dynamic range of approximately 10^{-38} to 10^{+38} . If range and accuracy are required, the 64-bit format spans in excess of 10^{+300} to 10^{-300} . This wide dynamic range, in conjunction with highly accurate rounding algorithms, renders the iSBX 332 module ideal for scientific problems and other applications requiring high speed, accuracy, and range. □

MULTIPROCESSING SYSTEM MIXES 8- AND 16-BIT MICROCOMPUTERS

Combining different single-board computers on a single bus is a promising way to achieve the most suited configuration for each task most suited to the task.

April 1980

Joseph P. Barthmaier, Intel Corporation, Hillsboro, Oregon

Multiprocessing System Mixes 8- and 16-Bit Microcomputers

Joseph P. Barthmaier
Computer Design, February 1980

MULTIPROCESSING SYSTEM MIXES 8- AND 16-BIT MICROCOMPUTERS

Combining different single-board computers on a single bus and assigning to each the tasks most suited enable a cost-effective multiprocessing system configuration with improved throughput and reliability

Joseph P. Barthmaier

Intel Corporation, Hillsboro, Oregon

Two or more single-board computers can share a common system bus to provide improved performance, reliability, and cost-effectiveness in medium to large scale applications. Interfacing multiple computers across a system bus affords a dual-bus architecture in which global system traffic is isolated from local traffic on the board buses. This allows a straightforward design of modular multiprocessing systems that combines different computer boards, and allocates to each that portion of the overall system function to which it is best suited.

In a typical design, 8- and 16-bit single-board computers (SBCs) communicate across a system bus to service an application that requires both realtime data acquisition and extensive signal processing. Partitioning system tasks and assigning each to the appropriate SBC optimizes performance without adding components. Dual-port memory provides a convenient way to synchronize

processes on different SBCs. Because most system functions are isolated on one SBC, reliability and throughput are increased, and implementation is facilitated.

Single-Board Computer Concept

In earlier SBC design, the fundamental goal was to provide a board containing all the resources required for a large variety of microprocessing applications. A typical processor board supplies an 8080A processor, 4k bytes of random access memory (RAM), sockets for up to 8k bytes of erasable programmable read only memory/read only memory (EPROM/ROM), a serial input/output (I/O) interface, 48 parallel I/O lines, three timer/counters, and eight levels of priority interrupt. With this hardware configuration, many small applications can be served with no need for additional memory or digital logic.

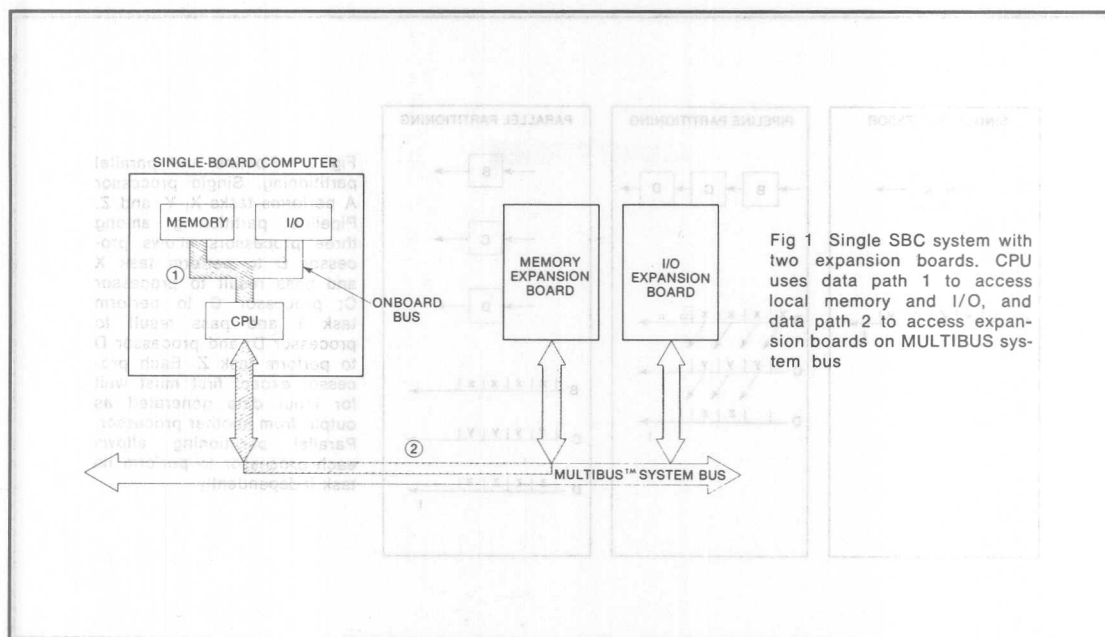


Fig 1 Single SBC system with two expansion boards. CPU uses data path 1 to access local memory and I/O, and data path 2 to access expansion boards on MULTIBUS system bus

Because larger applications require additional resources, an external bus structure was defined. The MULTIBUS™ system bus was designed for communication between SBCs and system expansion boards. Address, data, and handshake lines were defined for memory and I/O transfers between SBCs and expansion boards. There are bus expansion boards for system expansion in areas of RAM and ROM storage, serial and parallel I/O, analog I/O, and peripheral controllers.

In Fig 1, two buses interconnect a system with an SBC and two expansion boards. An onboard bus accesses local resources, and the system bus accesses global resources. A key advantage of this structure is that an SBC may not require the system bus for a large portion of its memory or I/O transactions. In many applications, less than 10% of the time is taken by system bus accesses. The large amount of potential system bus capacity makes this architecture a natural candidate for multiprocessing applications. As additional SBCs are included in the system, the incremental amount of system bus bandwidth required is usually small.

Motivations for Multiprocessing

Certain system applications benefit from using more than a single SBC. Motivations for constructing multiprocessing systems with SBCs include:

Resource sharing. In a multiprocessing system designed around the resource sharing concept, two or more processor boards share a common resource, such as a high

speed mathematics board or a peripheral controller. These boards perform independent functions with no relationship to one another except for the shared resource. Low cost is the obvious motivation for using a resource sharing multiprocessing configuration. If two processor boards share the same diskette controller, for example, overall system costs are considerably reduced.

Enhanced system throughput and performance. In many applications, significant improvements in performance may be achieved by using more than one processor in the system. Two ways of allocating or partitioning system functions among multiple processors, such as pipeline and parallel partitioning, are shown in Fig 2. In pipeline partitioning, system functions (tasks) are divided among several processors, so that data flow through the system is primarily serial. Each processor performs its portion of system functions, and then calls upon another processor to perform another set. An example of pipeline partitioning is when one processor performs data acquisition and buffering, while a second uses the data to perform digital signal processing.

Parallel partitioning allocates system functions among several processors in such a way that each processor performs a separate system task in parallel. An example is a system where one processor performs an industrial process control loop, while another monitors and controls a varying parameter, such as temperature.

Few systems may be characterized as totally parallel or pipeline partitioned, but designating systems in this manner can often be helpful during the system design phase, particularly when interprocessor communication software is being designed.

Modularly configured systems. A primary design goal, particularly in systems that are produced in low volume,

MULTIBUS is a registered trademark of Intel Corp.

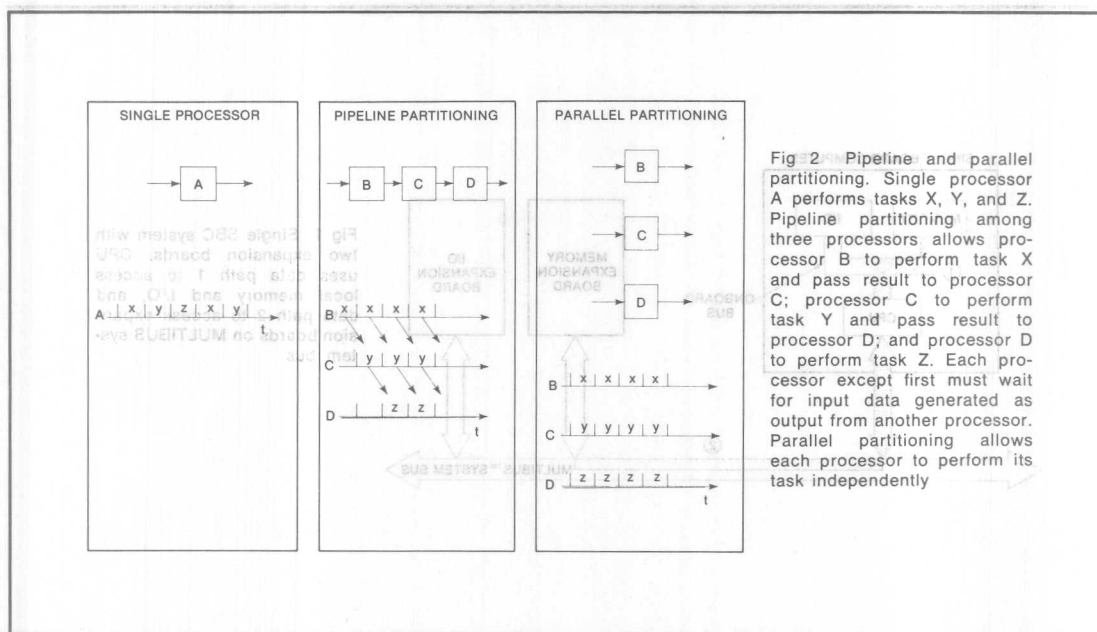


Fig. 2 Pipeline and parallel partitioning. Single processor A performs tasks X, Y, and Z. Pipeline partitioning among three processors allows processor B to perform task X and pass result to processor C; processor C to perform task Y and pass result to processor D; and processor D to perform task Z. Each processor except first must wait for input data generated as output from another processor. Parallel partitioning allows each processor to perform its task independently.

is often flexibility of system configuration. Using modularly configured systems, independent hardware and software modules are designed and implemented with individual processors or intelligent slave boards. When a particular configuration is required, the system designer selects the necessary hardware and software modules and combines them with interprocessor communication software. Shortened system development time, simple debugging, and a convenient upgrade path for system expansion are the benefits of such a technique.

High reliability. Multiprocessing may be used to isolate system tasks on individual processors in applications where a high degree of reliability is a requirement. If a processor fails, the remainder of the system continues to operate. Redundant designs, where a second processor may be dynamically assigned to perform the functions of a disabled processor, are a possibility.

Multiprocessing With Single-Board Computers

The MULTIBUS architectural design facilitates multiprocessing because multiple bus masters are accommodated, bus masters generate and acknowledge bus interrupts, and dual-port memory and intelligent slave architecture can be implemented.

Multiple Bus Masters

A bus master is a dynamic board that takes control of the bus by asserting address and control lines. Only one bus master may control the bus at any given moment. Examples of bus masters include single-board computers

and direct memory access (DMA) controllers. A bus slave is a passive element on the bus that does not assert address and control lines. Examples of bus slaves include memory or I/O expansion boards and intelligent slaves.

Several control lines exist on the bus so that potential bus masters can exchange control. These control lines, plus logic on the master boards, implement a priority scheme in which the highest priority master requesting the bus obtains control. There are two priority resolution schemes for exchange of the bus, serial and parallel. Using serial priority resolution, there may be up to three bus masters in the system; the parallel technique allows up to 16. A bus master is always given the opportunity to complete a bus transaction before being preempted by a higher priority master. In addition, bus masters may retain control of the bus by "locking" the bus. The bus lock feature is required when a master must have exclusive control of the bus for such functions as testing and setting software semaphores and completing operations involving I/O devices.

Since SBCs have extensive onboard resources, system bus transactions are not required for all I/O and memory accesses. Depending on the application and system design, multiple bus master systems with a small number of transactions can be configured. The system design goal is to use onboard resources whenever possible. Frequently executed or time critical code should be stored in onboard memory to minimize system bus accesses and to avoid delays while contending for the bus.

Interprocessor Interrupts

Eight interrupt lines exist on the system bus. In addition to interrupts from I/O slave boards or DMA controllers,

these interrupt lines can be used for communication between master SBC boards. Individual master boards may either generate interrupts or be interrupted from one or more of the interrupt lines. Interprocessor interrupts provide a fast and effective way for multiple SBCs to communicate over the system bus.

Dual-Port Memory

Single-board computers have been designed with onboard RAM containing two access ports. Dual access ports permit the onboard CPU to access the RAM directly using the onboard bus. Other SBCs also access the RAM using the system bus. The amount of memory available for system bus access may be selected from all memory accessible to no memory accessible, in increments of one-half or one-quarter of available memory size. This ability to block RAM access from the system bus provides memory protection for data and code stored in those nonaccessible areas of the dual-port RAM. Fig 3 illustrates an example of two SBCs accessing the dual-port memory of one SBC.

Two important benefits are gained by using the dual-port architecture. First, in a multiple-processor system, if two processors communicate through shared memory, only one must access the memory using the system bus, and the amount of system bus traffic may be significantly reduced. Second, in a multiprocessor configuration where limited RAM storage is required, a separate memory board is not needed. Such small systems have all the required system bus-accessible memory on one or more of the SBCs.

Intelligent Slave Architecture

To distribute intelligence in larger systems, the intelligent slave concept was developed. An intelligent slave is a

board that contains a CPU, some dedicated I/O capability, and a dual-port RAM for interfacing to the system bus. For example, the iSBC 544™ intelligent communications controller contains an 8085A processor, four 8251A serial I/O devices universal synchronous/asynchronous receiver/transmitters (USARTs), 12 levels of priority interrupt, and 16k of dual-port RAM. All communication between a master processor board, such as an iSBC 86/12A™ board, and the 544 takes place using the 544 dual-port RAM (Fig 4). The 8085A processor does not have the capability of taking control of the system bus (becoming a bus master) and accessing other system resources.

The 544 board was designed to operate using only onboard resources. The master SBC in the system transfers blocks of data and parameters to or from the 544 using the onboard dual-port RAM. To facilitate communication with the 544, an interrupt occurs when a master SBC writes into the lowest byte of memory of the dual-port RAM. The intelligent slave board can interrupt a master SBC by asserting one of the system bus interrupt lines with an I/O instruction. The address space occupied by the dual-port RAM may be set anywhere within 1M bytes; 20 address bits are decoded.

Primary advantage of intelligent slave architecture is the ease with which multiprocessing applications may be implemented. The intelligent slave may be sent a buffer of data and commands with an interrupt occurring, via a write to the lowest byte of memory, as a start command. The master SBC may continue operation with other functions to be notified, via an interrupt or a status byte in dual-port RAM, when the slave has completed a task. Since the intelligent slave may not access system resources via the system bus, no interference with the master SBC can occur.

isbc and the combination of isbc and a numerical suffix are registered trademarks of Intel Corp

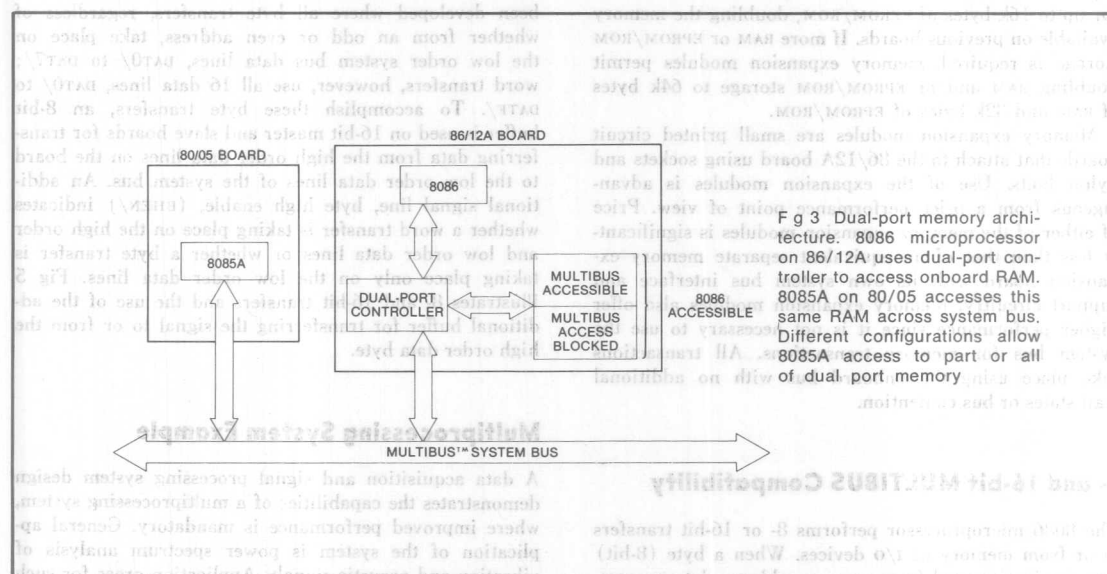


Fig 3 Dual-port memory architecture. 8086 microprocessor on 86/12A uses dual-port controller to access onboard RAM. 8085A on 80/05 accesses this same RAM across system bus. Different configurations allow 8085A access to part or all of dual port memory

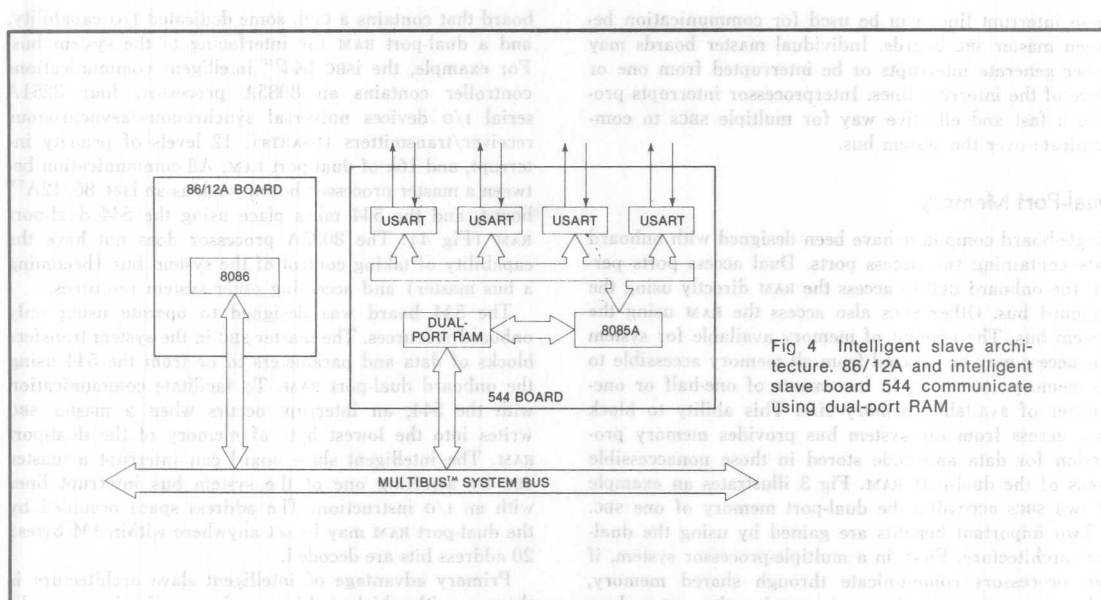


Fig 4 Intelligent slave architecture. 86/12A and intelligent slave board 544 communicate using dual-port RAM

The 86/12A Board

The iSB 86/12A single-board computer¹ has many of the architectural features of 8-bit boards (serial and parallel I/O, multiple interrupt levels, and timer/counters) but includes a 5-MHz 8086 microprocessor and larger amounts of RAM and EPROM/ROM storage. The 16-bit 8086 permits byte and word transfers, hardware multiply/divide, 1M-byte addressability, extensive string manipulation instructions, and many other features. The 86/12A contains 32k bytes of dual port RAM and sockets for up to 16k bytes of EPROM/ROM, doubling the memory available on previous boards. If more RAM or EPROM/ROM storage is required, memory expansion modules permit doubling RAM and/or EPROM/ROM storage to 64k bytes of RAM and 32k bytes of EPROM/ROM.

Memory expansion modules are small printed circuit boards that attach to the 86/12A board using sockets and nylon bolts. Use of the expansion modules is advantageous from a price/performance point of view. Price of either of the memory expansion modules is significantly less than that of an equivalent separate memory expansion board with its own system bus interface and support circuitry. Memory expansion modules also offer higher performance since it is not necessary to use the system bus for memory transactions. All transactions take place using the onboard bus with no additional wait states or bus contention.

8- and 16-bit MULTIBUS Compatibility

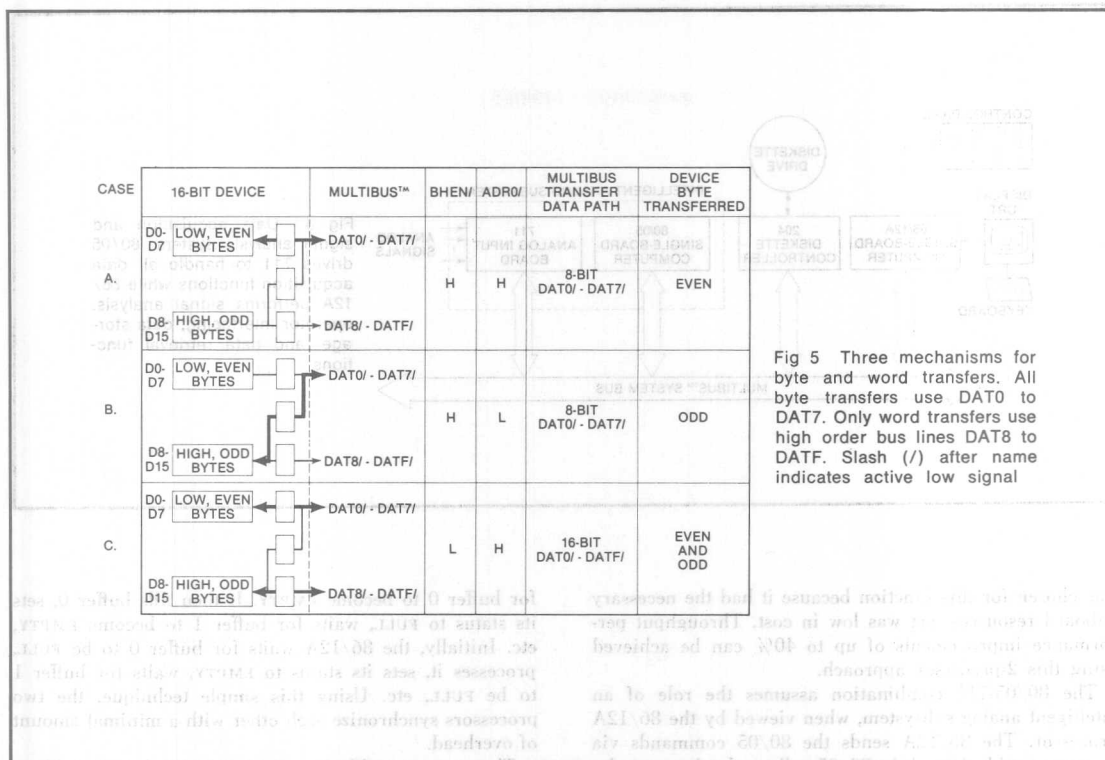
The 8086 microprocessor performs 8- or 16-bit transfers to or from memory or I/O devices. When a byte (8-bit) transfer is requested from an even address, data are pre-

sented to the microprocessor on its low order data lines, D0 through D7. When a byte transfer is requested from an odd address, data transfer must occur on the high order data lines, D8 through D15. When a 16-bit (word) transfer is requested, data are transferred on all 16 data lines, D0 through D15. When an 8-bit microprocessor (8080A or 8085A) is used, however, all byte transfers must take place on data lines D0 through D7, the only lines available.

To maintain compatibility between boards with 8-bit and 16-bit processors, a system bus transfer protocol has been developed where all byte transfers, regardless of whether from an odd or even address, take place on the low order system bus data lines, DAT0/ to DAT7/; word transfers, however, use all 16 data lines, DAT0/ to DATF/. To accomplish these byte transfers, an 8-bit buffer is used on 16-bit master and slave boards for transferring data from the high order data lines on the board to the low order data lines of the system bus. An additional signal line, byte high enable, (BHEN/) indicates whether a word transfer is taking place on the high order and low order data lines or whether a byte transfer is taking place only on the low order data lines. Fig 5 illustrates 8- and 16-bit transfers and the use of the additional buffer for transferring the signal to or from the high order data byte.

Multiprocessing System Example

A data acquisition and signal processing system design demonstrates the capabilities of a multiprocessing system, where improved performance is mandatory. General application of the system is power spectrum analysis of vibration and acoustic signals. Application areas for such



systems include vibration analysis in mechanical structures such as electric motors, automobiles, aircraft, and buildings, as well as speech, sonar, and low frequency radar analysis.

Design objective is to monitor the condition of large electric motors. Power spectra of vibration signals from various points on the motor are calculated in order to detect bearing wear and to predict an impending motor failure. Calculated power spectra are compared with reference spectra, and, if thresholds in various regions of the spectra are exceeded, an operator alarm is activated. Information regarding the state of the motors and the reference spectra is stored on disc.

The system monitors 16 channels of analog input signals generated by pairs of accelerometers mounted on each of eight motors. Sampling and calculations for the two channels of a single motor are performed simultaneously; then the next motor in sequence is monitored.

Fast Fourier transform (FFT) of a buffer of samples from an analog to digital converter (ADC) performs power spectrum calculations. Real and imaginary parts of the FFT results are squared and summed to form a power spectrum that is compared to the reference power spectrum in order to determine if the motor vibrations are within acceptable tolerances. A CRT displays calculated and reference power spectra. At periodic intervals, data are stored on disc for archiving the condition of each of the motors. If the motor spectra exceed the reference

alert the operator.

System Hardware

As shown in Fig 6, the 711 analog input board, containing a 12-bit ADC, samples the 16 analog signals from the motors. The 80/05 processor board drives the 711 analog board and handles all system data acquisition activities. The 80/05 contains an 8085A CPU, 512 bytes of RAM, up to 4k bytes of EPROM/ROM, a timer/counter, parallel and serial i/o lines, and four levels of priority interrupt. The 86/12A board is the main system processor. The 8086-based board performs all the signal processing functions, displays the spectra on the CRT, drives the system control panel, and transfers motor condition data onto disc using the 204 single-density diskette controller.

Increased system performance is the design motivation for using two processor boards. The 86/12A board, with its 5 MHz 8086 CPU, 16-bit multiply/divide capability, 64k bytes of dual-port RAM, and 32k bytes of EPROM/ROM, is used for the mathematically intensive power spectrum calculations.

The 80/05 processor board is used to offload data acquisition activities from the main processor. It assumes all the overhead of handling the 711 analog board. Sampling is performed at 250-μs intervals using the onboard timer; data from the two channels are scaled, demultiplexed, and stored in a buffer. The 8-bit processor board

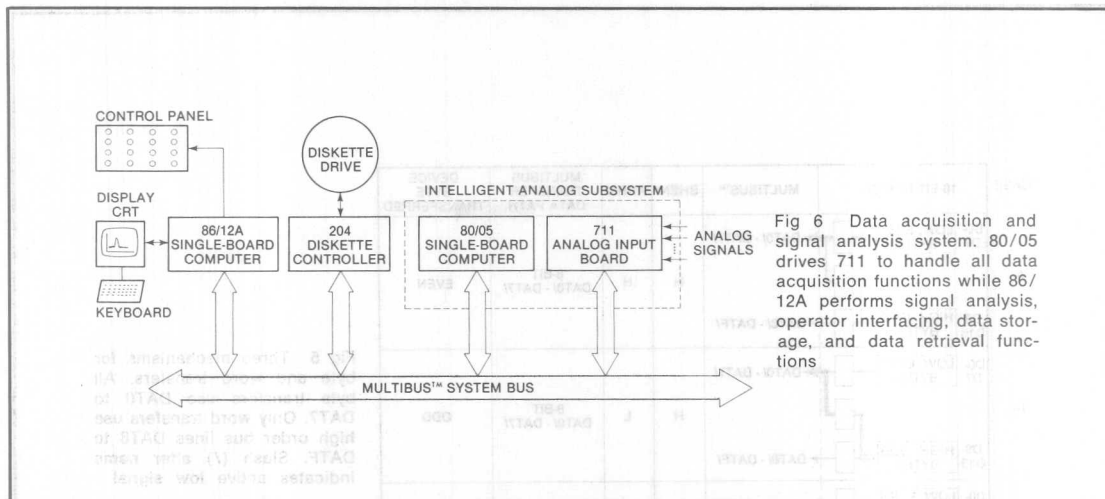


Fig 6 Data acquisition and signal analysis system. 80/05 drives 711 to handle all data acquisition functions while 86/12A performs signal analysis, operator interfacing, data storage, and data retrieval functions

was chosen for this function because it had the necessary onboard resources, yet was low in cost. Throughput performance improvements of up to 40% can be achieved using this 2-processor approach.

The 80/05-711 combination assumes the role of an intelligent analog subsystem, when viewed by the 86/12A processor. The 86/12A sends the 80/05 commands via a parameter block, and the 80/05 collects the data samples in buffers. When a buffer is complete, the 80/05 signals the 86/12A using the parameter block. Thus, the 80/05 acts as an intelligent DMA controller for the 711 board.

System Software

Due to the large RAM requirements of the system, the ISBC 300™ RAM expansion module is used to increase RAM capacity to 64k bytes. Memory has been configured to make 16k bytes of memory accessible to the system bus, with the remaining 48k bytes reserved for use by the onboard 8086 and not accessible to the system bus. The amount of 86/12A dual-port RAM that is system bus accessible may be configured in 16k increments from zero (no memory accessible) to 64k (all memory accessible). The parameter block used for interprocessor communication and a pair of buffers used for storage of the analog samples are stored in the memory accessible to the 80/05. Memory not accessible to the 80/05 contains the data and buffers used for the calculated averaged power spectra, reference spectra, CRT displays, and disc data. The 16 bytes of the parameter block contain all information required for communication between the two SBCs in the system, including buffer addresses, status, size, sample rate, and start and end channel.

Fig 7 is a flow diagram illustrating how buffer status bytes are used to synchronize the filling and processing of the data buffers. Each buffer may be in one of two states, FULL or EMPTY. Initially, both buffers are EMPTY. At initialization, the 80/05 fills buffer 0, sets its status to FULL, fills buffer 1, sets its status to FULL, and waits

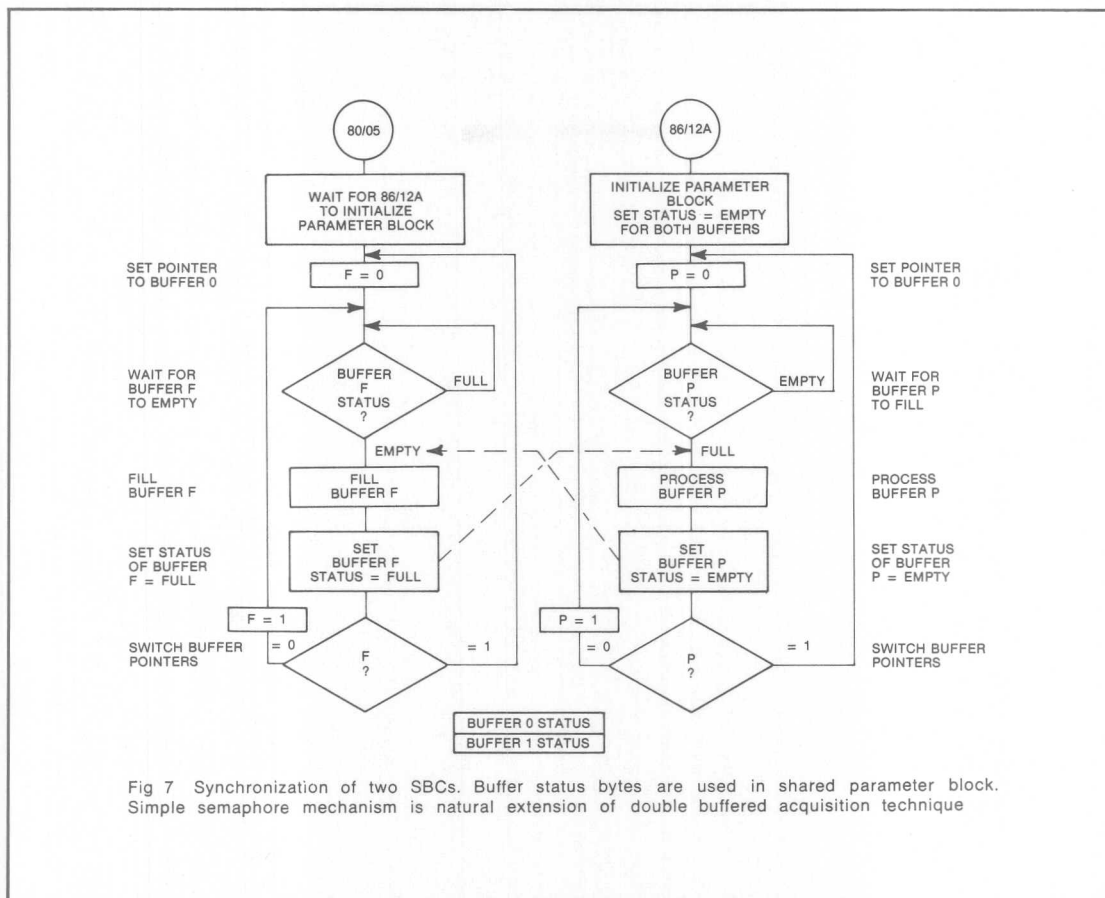
for buffer 0 to become EMPTY. It then fills buffer 0, sets its status to FULL, waits for buffer 1 to become EMPTY, etc. Initially, the 86/12A waits for buffer 0 to be FULL, processes it, sets its status to EMPTY, waits for buffer 1 to be FULL, etc. Using this simple technique, the two processors synchronize each other with a minimal amount of overhead.

The parameter block approach is used to provide a simple means for interfacing the two SBCs. At system initialization, the 80/05 board needs only to know the base address of the parameter block. Once this is known, all other information required for the 80/05 to function properly is available. The end application and even the specific type of SBC that calls upon the 80/05 for data samples remain irrelevant to the 80/05. Driver software for the 80/05 is therefore highly modular and may be used in a variety of applications and configurations with no changes required.

A key capability of this system design is that the 86/12A board does not use the system bus to access data samples, thus minimizing execution time for the highly iterative FFT computation. The 80/05 processor takes the samples from the analog board and stores them directly into the 86/12A dual-port memory. Therefore, except for occasional disc transfers by the 86/12A, the 80/05 is the only processor using the system bus. This increases system throughput and eliminates contention for the system bus.

Signal Processing Software

The algorithm used for the FFT in this application is known as "time decomposition with input bit reversal."² Using this algorithm, an in-place FFT has been programmed for an input frame size of 128 complex points. Sixteen-bit integer mathematics is used for all internal calculations of the FFT. The 86/12A board computes the 128-point complex FFT in 110 ms. Computation of the averaged power spectra is performed using a double pre-



cision integer format. The 16-bit integer real and imaginary values which result from the FFT are squared and summed to obtain a 32-bit power spectrum. Thirty-two frames of data are processed and summed to form the averaged power spectrum.

Summary

Two reasons for the slow growth of multiprocessing have been the limited selection of SBCs and the relatively small application base. These conditions are changing rapidly due to the large number of SBCs now available. These boards contain dual-port RAM and newer 8- and 16-bit CPUs, and provide system designers with a comprehensive set of tools for tackling applications that require the power of multiprocessing. Thus, the SBC application base has grown significantly in recent years.

The system application combines a low cost 8-bit SBC and a high performance 16-bit SBC in a configuration designed for both data acquisition and signal analysis. The 8-bit SBC relieves the 16-bit SBC of all system data acquisition functions. Because the 16-bit board spends

full time processing data, system throughput can be increased by as much as 40%.

References

1. *iSBC 86/12A Hardware Reference Manual*, 98003074-01, Intel Corp, Santa Clara, Calif, 1979
2. S. D. Stearns, *Digital Signal Analysis*, Hayden Book Co, Rochelle Park, NJ, 1975



Joseph P. Barthmaier is applications engineering manager for the OEM Micro-computer Systems Operation at Intel. He is responsible for application notes and seminars covering the SBC product family and for future product planning. He holds BSEE and MSEE degrees from the University of Florida and Stanford University, respectively.

October 1977

RMX/80™ Real-Time Multitasking Executive

Thomas Rolander
OEM Microcomputer
Systems Applications

RMX/80 Real-Time Multitasking Executive

Contents

INTRODUCTION	2-5
OVERVIEW	2-5
Basic Concepts	2-5
General Characteristics	2-5
Nucleus Operations	2-7
EXTENSIONS	2-10
Free Space Manager	2-10
Terminal Handler	2-11
Disk File System	2-11
Debugger	2-11
USING THE RMX/80™ SOFTWARE	2-11
Task and Exchange Definition	2-12
Priority Assignment	2-13
Static Descriptions	2-13
Compilation/Assembly	2-13
Linking	2-14
Locating	2-14
Debugging	2-14
APPLICATIONS	2-14
Minimal Terminal Handler	2-14
Closed-Loop Analog Control	2-18
APPENDIX A	2-21
APPENDIX B	2-25
APPENDIX C	2-27

INTRODUCTION

A large number of microcomputer applications require the ability to respond to events in real time. RMX/80 provides the system software around which you can build a real-time multitasking application on Intel SBC 80 Single Board Computers. In addition, RMX/80 increases the utilization of a Single Board Computer by allowing its resources to be shared among several tasks executing concurrently. Synchronization of these multiple real-time tasks is handled by RMX/80, freeing you to concentrate your major programming efforts on your application.

This application note begins with an overview of RMX/80. Readers who are familiar with the material presented in the RMX/80 User's Guide may choose to skip to the next section, a description of how to use RMX/80 and the steps involved in using it by describing two applications.

- An interrupt driven minimal terminal handler for a CRT or Teletypewriter.
- A closed-loop analog control subsystem utilizing the Intel SBC 711 analog-to-digital board.

Each example has diagrams illustrating the relationships between its tasks and exchanges. These are useful tools in conceptualizing the activities taking place in real time. Program listings of the applications are interspersed with text describing the application.

OVERVIEW

Real-time systems provide the ability to control and respond to events occurring asynchronously in the physical world. Later in this application note, a process control application is described that monitors and controls the temperature within several chambers. The system controls the process by simply turning on and off a heat source. The system could also display the temperature on an operator's console and permit entry of new set-point temperatures and error ranges.

A single large program could have been used to perform the functions in a sequential manner. However, this approach may not permit an operator to enter control variables at the same time the process is being monitored and controlled. In contrast, real-time systems do not operate sequentially. A number of events may all be happening at the same time. This concurrence of events is a distinguishing characteristic of real-time systems.

BASIC CONCEPTS

There are basically three concepts that the user must master to effectively use RMX/80. The first is the task, an independent program which competes for resources within the system. The second concept is the message. Messages convey data and synchronization information between tasks. The third concept is the exchange. An exchange enables one task to send a message to another. As we will see later, the interaction between tasks and exchanges enables the user to implement mutual exclusion, communication, and synchronization. Mutual exclusion is a technique that controls access to a shared resource such as an I/O device or a data structure.

Task

Under RMX/80, the user codes a separate program, known as a task, for each event. An arbitrary number of these tasks execute concurrently and are subject to synchronization as required by their functions. Tasks share resources such as data structures and can communicate between themselves.

Message

A message is a unit of communication between tasks. Together with the exchange mechanism, it conveys information between tasks and can synchronize their operations.

Exchange

RMX/80 uses message exchanges for task-to-task communication. An exchange is a pair of queues represented by a data structure at which messages are left by one task to be picked up by another. Tasks may send messages to an exchange, and may wait for messages at an exchange. A task which waits for a message may perform a timed or an untimed wait. A timed wait will terminate upon the receipt of a message or at the end of the specified period of time, even if it has not received a message. When a task does an untimed wait for a message it is guaranteed that the task will not execute again until a message is available for it. A representation of the exchange data structure is shown in Figure 1.

GENERAL CHARACTERISTICS

In addition to the basic concepts of tasks and exchanges, several other general characteristics of RMX/80 are relevant in this overview.

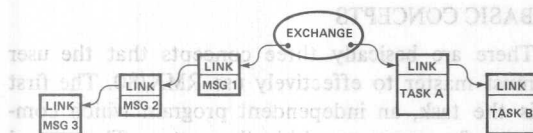


Figure 1. Exchange Data Structure

System Time Unit

RMX/80 uses a system time unit that is the period of time between "ticks" of the system clock. The standard RMX/80 system time unit is 50 milliseconds. The system time unit provides timing and user task scheduling. A task may wait at an exchange for a specified number of system time units and then continue execution. A task could be written to generate messages at specific time intervals. Tasks waiting for the messages would then be scheduled according to those time intervals.

Message Producing/Consuming Tasks

In general, tasks can be classified as message producing or message consuming tasks. The processing flow of these types of tasks are usually cyclic in nature and can be shown as follows.

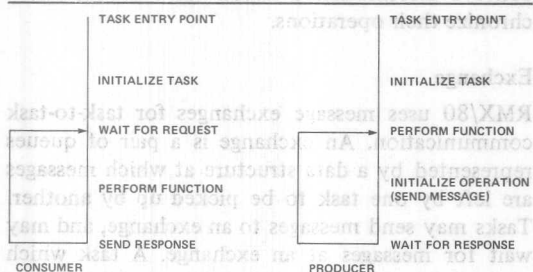


Figure 2. Message Producing/Consuming Tasks

A consumer task waits for a message to be posted at a particular exchange and takes control of the processor only when it has received a message and no other tasks of higher priority are ready to execute. The consumer task performs some action based upon the message and then simply resumes waiting until the next message is received. Usually, the consumer task acknowledges completion of its function by sending a response message to some other exchange associated with a task.

A producer task initiates its function by sending a message to another exchange and then surrenders control of the processor. The task continues to wait until it receives a response to its message.

Notice that the distinction between these types of tasks is relative since most tasks both produce and consume messages. However, the producer/consumer concept helps clarify the general structure of tasks—tasks are typically programmed loops. A producer task performs a function, sends a message, waits for a response, then loops back to begin again. A consumer task waits for a message, performs a function, sends a response, then loops back to wait again.

Interrupts

Hardware interrupts are treated as messages from peripheral devices for which a task can wait, as if the interrupt were a message from some other task. These messages arrive at particular exchanges, called interrupt exchanges, but are otherwise treated as described above. The system provides the ability to mask particular interrupts so that no messages ever arrive at a particular interrupt exchange associated with the masked interrupt. In the event that the overhead associated with turning an interrupt into a message is too high, the interrupt can be treated by the user directly via a user supplied interrupt service routine.

Task States

Tasks may exist in a number of states. A task is *running* if it actually has the processor executing instructions on its behalf. A *ready* task is one that could be running (any wait for a message or time period has been satisfied), but a higher priority task is currently running. A task is *waiting* if it cannot be ready or running because it is waiting at an exchange for a message. A *suspended* task is one that is not permitted to run or compete for system resources until it is resumed. The relationships between the task states are illustrated in Figure 3.

Priority

Each task has associated with it a priority that indicates its importance relative to other tasks in the system and relative to the interrupts of peripheral devices. RMX/80 schedules a task for execution based on the task's priority. Whenever a decision must be made on which task should be

run, the highest priority ready task is chosen. Each of the eight hardware interrupt levels has a set of priorities, one of which must be assigned to the task that services the interrupt. When an interrupt occurs that task is executed if it is the highest priority ready task. At the time a higher priority task preempts a lower priority task, RMX/80 saves all the relevant information about the pre-empted task so it can eventually resume execution as though it were never interrupted. This process is known as a context save.

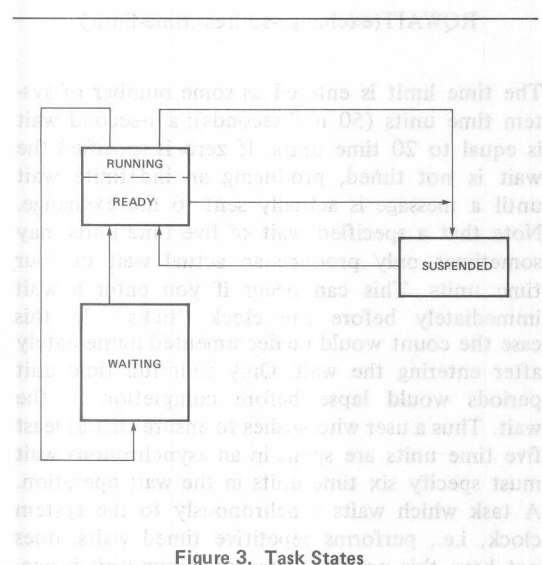


Figure 3. Task States

NUCLEUS OPERATIONS

The RMX/80 nucleus provides several operations that you can access with programmed calls. Two basic operations are covered in this section (additional operations are described in the RMX/80 User's Guide):

- RQSEND, send a message to an exchange
- RQWAIT, wait for a message or time interval

These two operations provide the capability to pass messages between tasks in a system running under RMX/80.

Message Format

The messages used by the send and wait operations to convey information between tasks are variable in length and contain the information shown in Figure 4.

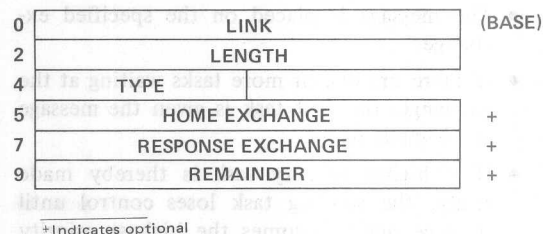


Figure 4. Message Format

Fields

1. LINK — a 2-byte field used to enter the message on a linked list at an exchange.
2. LENGTH — a 2-byte field containing the total length of the message in bytes. The minimum message length is 5 bytes (LINK, LENGTH, and TYPE).
3. TYPE — a 1-byte field indicating the type of message.
4. HOME EXCHANGE — an optional 2-byte field containing the address of an exchange to which this message should be sent when it has no further use. This field is very useful in implementing and managing a pool of messages.
5. RESPONSE EXCHANGE — an optional 2-byte field containing the address of an exchange to which a logical response to this message should be sent. This field is intended to specify the exchange at which a sending task is waiting for an acknowledgement message if one is needed.
6. REMAINDER — an optional field of arbitrary length that may contain any data portion of the message.

Sending a Message to an Exchange

The RQSEND operation enables a task to post a message at an exchange. When you send a message to an exchange, RMX/80 actually posts only the address of the message at the exchange, not the body of the message. RMX/80 avoids the overhead required to move an entire message to an exchange. Thus it is possible to queue a number of messages at the same exchange with little overhead in either execution time or memory requirements. When a task sends a message to an exchange, several functions are performed.

- The message is placed on the specified exchange.
- If there are one or more tasks waiting at the exchange, the first task is given the message and is made ready.
- If a higher priority task is thereby made ready, the sending task loses control until it once again becomes the highest priority ready task.

After a message is sent to an exchange, it must not be modified by the sending task. A task which then receives the message by waiting at the exchange where the message has been posted is free to modify the message. The format of the RQSEND operation is as follows.

RQSEND(exchange-address,message-address)

Message exchanges are defined by the user, and are normally addressed symbolically. For example, the exchange used to pass readings from an analog-to-digital (A/D) task might be named ATODEX. The reading itself could be contained in a message with the name RDNG. Thus, a typical call for a send in a PL/M program might be as follows:

```
CALL RQSEND (.ATODEX.,RDNG);
```

The call procedure in assembly language is as follows.

```
LXI    B,ATODEX
LXI    D,RDNG
CALL   RQSEND
```

The assembly language rules for passing parameters to RMX/80 are the same as for passing parameters to a PL/M procedure called from an assembly language module. For 2-byte parameters, the first parameter is passed in the B and C registers; the second parameter is passed in the D and E registers.

Waiting for a Message or Time Interval

The RQWAIT operation causes a task to wait for a message to arrive at an exchange. It is also possible to delay execution of a task when no message is anticipated for the task. The task simply waits for the desired time period at a message exchange where no message is ever sent. When a task waits for a message at an exchange several operations are performed.

- The task is made to wait until a message is sent to the specified exchange, or until the time limit has expired.
- When a message is available, its address is returned to the task.
- If the time limit expires before a message becomes available, a system TIMEOUT message is returned to the task.

The format of the RQWAIT operation is as follows.

RQWAIT(exchange-address,time-limit)

The time limit is entered as some number of system time units (50 milliseconds); a 1-second wait is equal to 20 time units. If zero is specified the wait is not timed, producing an indefinite wait until a message is actually sent to the exchange. Note that a specified wait of five time units may sometimes only produce an actual wait of four time units. This can occur if you enter a wait immediately before the clock "ticks." In this case the count would be decremented immediately after entering the wait. Only four full time unit periods would lapse before completion of the wait. Thus a user who wishes to ensure that at least five time units are spent in an asynchronous wait must specify six time units in the wait operation. A task which waits synchronously to the system clock, i.e., performs repetitive timed waits, does not have this problem because a new wait is executed following a tick that satisfied the previous wait. The following are typical calls for the RQWAIT operation.

PL/M

```
PTR = RQWAIT(.ATODEX,20);
```

The RQWAIT procedure returns an address value which is the address of a message.

Assembly Language

```
LXI    B,ATODEX
```

```
LXI    D,20
```

```
CALL   RQWAIT
```

The address of a message is returned in the HL register pair.

Send – Wait Interaction

To a large extent, the power of RMX/80 as a programming tool is derived from the interaction between send and wait. The interaction includes three multi-tasking operations.

- Communication
- Synchronization
- Mutual Exclusion

In describing these operations, a graphic notation for diagramming tasks, exchanges, and their interaction (send and wait operations) is useful. The notation is described in the next section on communication.

Communication. The most common interaction between tasks is that of communication – the transmission of data from one task to another via an exchange (Figure 5).

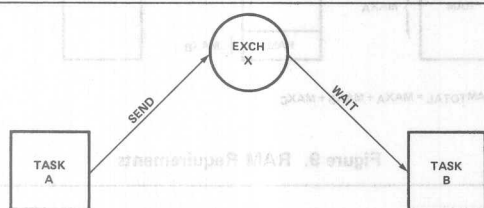


Figure 5. Communication

Rectangles designate tasks while circles represent exchanges. Arrows that are directed from tasks to exchanges indicate send operations. Wait operations are shown by arrows directed from exchanges to tasks.

Figure 5 shows an example of communication between task A and task B. Task A sends a message to exchange X and task B waits for a message at that exchange. Task A is the message producer and task B the message consumer.

Synchronization. At times there is a requirement to send a synchronizing signal from one task to another. This signal can take the form of a message that contains only header information, that is, LINK, LENGTH, and TYPE.

Let us consider the implementation of a task scheduler, used for the purposes of synchronizing another task that performs a particular function at periodic intervals. The relationship between the tasks and exchanges is shown in Figure 6.

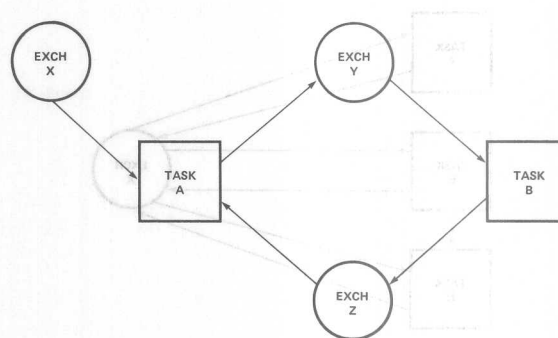


Figure 6. Synchronization

Task A, the scheduler, performs a timed wait on the X exchange. Note that the full wait period will always occur because there is no task that is sending messages to exchange X. In this manner, a specific timed wait by task A precedes the passing of a synchronization message from task A to task B via exchange Y, and then the return from task B to task A via the Z exchange.

If task B waited on X directly, rather than using task A for scheduling, it would be scheduled n system time units from when it waits instead of n units from the last time it was awakened. A comparison between the two methods is shown in Figure 7.

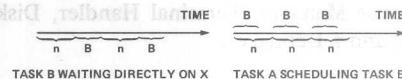


Figure 7. Scheduling Methods

Mutual Exclusion. In an environment with multi-tasking, resources often must be shared. Examples of shared resources include data structures and peripherals such as the Intel SBC 310 Math Module. Mutual exclusion can be used to ensure that only one task has access to a shared resource at a time. Figure 8 shows how an exchange can be used to limit access to a resource.

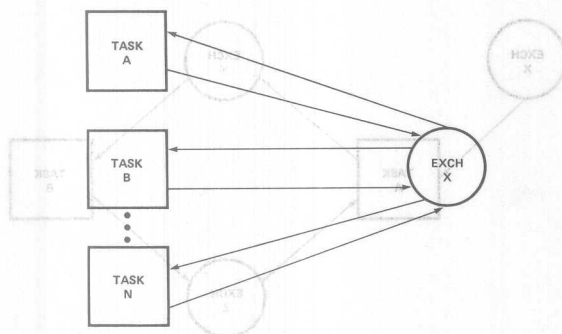


Figure 8: Mutual Exclusion

In this example, the X exchange is sent a single message at system initialization. Then, as tasks require the resource, they wait for a message from the X exchange. When the message is received, the task knows it has sole access to the resource because there is only one message associated with the exchange. After the task finishes with the resource it sends the message back to the X exchange. The next task waiting for the resource continues, knowing it has exclusive access to the resource.

EXTENSIONS

RMX/80 has several extensions which provide operations commonly used in real-time applications. The nucleus of RMX/80 requires less than two thousand bytes of memory and includes all of the basic operations. The extensions include a Free Space Manager, Terminal Handler, Disk File System, and a Debugger.

FREE SPACE MANAGER

The Free Space Manager maintains a pool of free RAM and allocates memory from that pool upon request from a task. The Free Space Manager also reclaims memory and returns it to the pool when it is no longer needed.

The Free Space Manager is especially useful in two applications. The first application arises from the need for variable length messages. If you have a task that produces messages of variable length, such as a task sending text for display on a CRT, the Free Space Manager can be used to provide a message to meet your exact size requirements.

An alternate solution is to maintain a pool of large fixed length messages. The pool can be maintained without the Free Space Manager; however, memory is wasted because of the unused space remaining in the fixed length messages.

The second application of the Free Space Manager relates specifically to effective use of memory. In a typical application, the total RAM requirement is computed by adding up the maximum RAM requirements for each task in a system as shown in Figure 9.

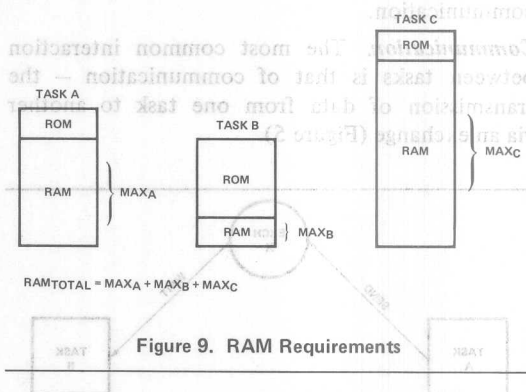


Figure 9: RAM Requirements

The efficiency of memory utilization is a function of the total RAM memory needed during typical system operation. Reducing the total amount of RAM by sharing it among the tasks often has little impact on total performance. However, significant cost advantages may be gained by reducing the total amount of memory. The memory requirements can be calculated as the minimum RAM for each task plus the pool (shared memory), as shown in Figure 10.

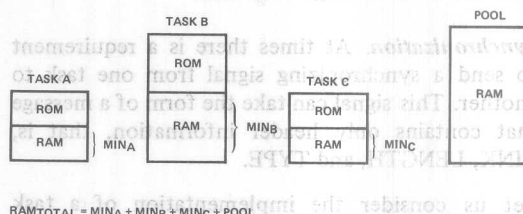


Figure 10: RAM Requirements Using a Pool

TERMINAL HANDLER

The Terminal Handler provides real-time asynchronous I/O between an operator terminal and tasks running under the RMX/80 executive. The Terminal Handler provides a line-edit capability similar to that of ISIS-II and an additional type-ahead feature. (ISIS-II is the diskette supervisor system used on the Intellec Microcomputer Development System.) Access to the Terminal Handler is provided by two exchanges where messages are sent to initiate read and write requests.

Several features of the Terminal Handler have been incorporated specifically to facilitate interaction with the debugger. Because of this interaction, the Terminal Handler is required for operation of the debugger.

DISK FILE SYSTEM

The Disk File System (DFS) provides users of RMX/80 with disk file management capabilities. This system allows user tasks to create, access, and maintain disk files in a real-time environment. This means that many I/O requests can be processed concurrently, rather than one at a time.

In addition to the file handling services, DFS provides a program loading facility that allows you to load program segments into memory from disk. The DFS can be configured to include only those functions which you require. For example, if your disk accesses are sequential rather than random, you omit the SEEK function. This philosophy of minimizing memory requirements by including only the functions your application requires is found in virtually all aspects of RMX/80.

DEBUGGER

An environment that is continually changing in response to asynchronous physical events can present a serious debugging challenge. The Debugger aids you in debugging tasks running under the RMX/80 executive. The Debugger provides a command language that can be used to passively display information about the system, or actively modify and interact with the system.

Passive Functions

Because RMX/80 manages a fairly complex set of data structures, the Debugger has the capability of displaying them in an intelligible format. The Debugger can be used in this manner to view tasks, exchanges, messages, and other data struc-

tures maintained within the RMX/80 environment. The contents of all RAM and ROM memory locations may also be displayed by the Debugger.

Active Functions

The active Debugger functions include those of modifying memory, setting breakpoints, and monitoring stack overflow. The memory modification commands enable you to update the contents of memory and to move a series of bytes from any location to any other location.

Breakpoints can be set, allowing you to gain control when encountered by a task. Two kinds of breakpoints are supported: execution breakpoints and exchange breakpoints. An execution breakpoint can be placed at any instruction within read/write (RAM) memory. When the breakpoint is reached, the task encountering the breakpoint is stopped from further execution. The task registers may then be examined and modified before resuming execution.

Exchange breakpoints can be used to detect RQSEND and/or RQWAIT operations performed on specified exchanges. The exchange breakpoint can thus enable you to monitor the activity of any of the exchanges in your system. The task executing the appropriate RQSEND or RQWAIT to an exchange which has a breakpoint is stopped, allowing you to examine the task. This enables you to breakpoint a ROM resident task. The breakpointed task and the message involved in the operation with the exchange may then be displayed and modified before resuming execution.

The debugger can also be used to monitor stack overflow. This function is provided by the Debugger SCAN command which examines the stacks of all tasks in the system at a specified periodic interval. The fact that each task's stack is initialized with a unique value allows stack overflow to be detected. When a task stack overflows, it is removed from the system and a message is displayed.

USING RMX/80

This section of the application note describes the steps involved in using RMX/80. The process begins with the definition of the individual tasks and exchanges in your application. It continues with a discussion of the data structures that you must prepare. The task coding, compilation or assembly, linking, and locating is also described.

Finally, some comments are directed towards debugging tasks within the RMX/80 environment.

Before the details of using RMX/80 are discussed, some general observations are necessary to determine the suitability of RMX/80 for your application. To effectively utilize RMX/80, your application must either use interrupts or require device polling. Thus, the key element is the need to respond to external events. If your application satisfies this criteria, it is a likely candidate. However, you must then determine if RMX/80 is capable of supporting your application. This can be done by examining your interrupt response time and frequency requirements. The time required to transform an interrupt into a message that is sent to an interrupt exchange is approximately 800 microseconds for an SBC 80/20. This is the RMX/80 interrupt latency. It can be reduced to 60 microseconds by handling the interrupt directly, using the RQSETV operation to bypass the RMX/80 interrupt exchange mechanism. In this latter mode, an interrupt-driven asynchronous block transfer rate of about 10 kHz can be achieved.

TASK AND EXCHANGE DEFINITION

The initial design step for an application that runs under the RMX/80 Executive is to define your tasks, exchanges, and the interaction between them. This is perhaps best accomplished using the graphic notation introduced earlier in the section on Send — Wait Interaction. The graphic notation provides a clear picture of the relationships between the tasks and exchanges in your system. You can begin either in a top-down or bottom-up fashion. That is, you can use a top-down approach to define, at a gross level the operation of your system and then gradually break it down to the individual tasks. Or, you can start with the tasks associated with the external events in your application and then build the pieces to form the gross structure of your system.

The bottom-up approach forces you to begin with external events that drive your system. The number of these events, the amount of processing required, and the relationships between them define the tasks and exchanges in a system. For example, consider a system that samples an analog input with an A/D converter. Assume that the A/D provides an interrupt at the completion of a conversion. To use the data from the A/D converter it may also be necessary to scale it and add an offset.

With this information the portion of the task and exchange definition that relates to this function can be constructed.

Begin with the external event, the interrupt from the A/D. An interrupt priority level must be assigned to the A/D converter. This same level will be used by the task which waits on the interrupt exchange.

The relationship between the interrupt exchange and the A/D task is shown in Figure 11. If processing must be performed on raw data from the A/D, a second, lower priority, task could be used. Another task for this function will require a synchronizing signal from the ADC task to indicate that raw A/D data has been obtained and is ready for processing.

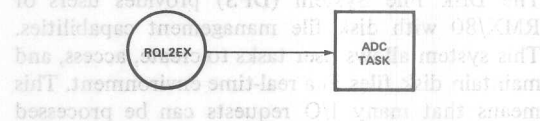


Figure 11. Interrupt Exchange and A/D Task

The interaction between the ADC task and the CONV task that processes the raw A/D data is shown in Figure 12. Two exchanges provide synchronization. The ADC task uses the TRGR exchange to signal that data is ready for processing by the CONV task. The CONV task uses the RTRGR exchange to signal the completion of its processing and thus its readiness to accept more raw data.

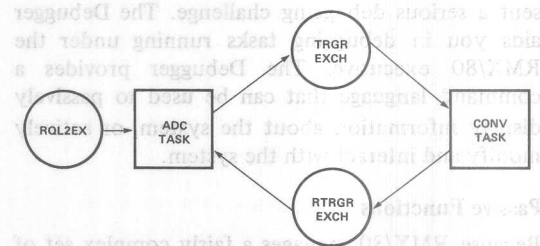


Figure 12. ADC and CONV Task Interaction

In this example two tasks and three exchanges have been defined. To develop an entire system, the tasks and exchanges associated with all of the external events in the system can be defined in the same manner. Then, proceeding bottom-up, the next step is to define the tasks and exchanges required to support the interaction between tasks running at the level of the real-time events.

After defining the entire application, you can begin actual coding of the tasks. You may choose to code in either assembly language or the PL/M 80 high-level language. Where possible, it is desirable to code in PL/M because PL/M lends itself to structured programming. Assembly language often encourages an ad hoc approach. Even if your application ultimately requires assembly language coding because of critical time and/or space parameters, initial design work in PL/M followed by translation into assembly language is recommended.

A total of 15 operations are supported by the RMX/80 nucleus. Only two of the operations, RQSEND and RQWAIT, are described in any detail in this application note. The remaining operations are described in the *RMX/80 User's Guide*. The reason for presenting only the send and wait operations is because they are sufficient for the implementation of a large number of real-time applications. These two operations provide a great deal of power and flexibility, yet their simplicity should enable those who are new to real-time programming to quickly develop applications.

PRIORITY ASSIGNMENT

The relative priority of tasks within a system running under RMX/80 determines which task is to be executed. Therefore, the assignment of a priority to each task is extremely crucial. For example, consider a compute bound task placed at a higher priority than an interrupt-driven task responsible for servicing an I/O device. This improper assignment of priorities could result in missed interrupts from the I/O device. Several steps can be followed in the assignment of task priorities.

1. Assign hardware interrupt priority levels according to the requirements of your application.
2. Specify priorities for the tasks which service the interrupts. These tasks should generally be short and serve only to perform the data

transfers. A second task with a priority lower than those assigned to the hardware interrupts should be used where further processing of the data is required.

3. Priority assignment should be made for all other tasks in the system based on the relative importance and interaction among the tasks.

Unfortunately the last step in assigning task priorities is largely intuitive. In fact, you may need some empirical data from actually running your application before you settle on your final task priority assignment.

STATIC DESCRIPTORS

When a system running under RMX/80 begins execution, several tables of data are used to initialize the system. These tables usually reside in ROM. The first table is the create table (RQCRTB) that specifies the number of tasks and exchanges in the system, and the addresses of the initial task table and the initial exchange table. The initial exchange table contains the addresses of all the exchange descriptors. The initial task table contains the static task descriptors for each task, and contains the following task parameters.

1. Name
2. Initial PC — the location at which to start task execution
3. Initial SP — the location at which to start the task stack
4. Stack length
5. Priority
6. Initial Exchange (described in the *RMX/80 User's Guide*)
7. TD Address — the RAM address of the task descriptor

You must prepare all three of these tables to produce a configuration module for RMX/80. The release diskette for RMX/80 includes a set of files which contain assembly language macros that simplify the preparation of your configuration module. The relationship between these tables is shown in Figure 13.

COMPILATION/ASSEMBLY

Preparing program segments for compilation and assembly can be simplified by use of files provided on the RMX/80 diskette. As described in the last

section, a set of macros is included to assist you in preparing your configuration module. Other files are provided that are useful when coding calls to RMX/80 and preparing data structures in PL/M.

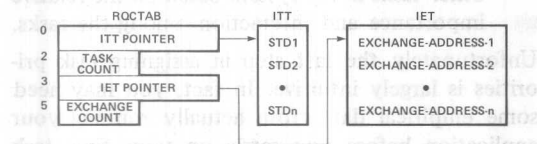


Figure 13. ROM Based Tables

By coding in a modular fashion you can separately compile and maintain tasks. This is advisable since a single large module containing all your tasks would require a lengthy recompilation to change any one of the tasks. Following the compilation and assembly of your source code modules, a library containing the object modules can be created.

LINKING

The process of linking prepares a single object module from libraries containing the RMX/80 object modules and your own application libraries or separate object modules. The order in which you specify the files to be linked is crucial to successful linking. In general, your libraries or separate object modules should be specified before the RMX/80 libraries. The link command should conclude with the unresolved library (UNRSLV.LIB) that contains miscellaneous modules for resolving PUBLICs not used in the application code. PUBLICs extend the scope of variables to allow linkage between separate program modules. Figure 14 illustrates how an application program is linked from RMX/80 and user tasks.

LOCATING

It is appropriate in this section to give some guidelines regarding the assignment of RAM and ROM address space for your Single Board Computer environment. The SBC 80 Single Board Computers have ROM based at location 0. Since the LOCATE program places all code in a contiguous block, the code must begin at location 0. Likewise, the read/write (RAM) data is also placed in a contiguous block. The base address of data should be placed at your RAM base address. Depending on the

amount of code space required by your application it may be necessary to move the RAM memory base address on your SBC to a higher location. A STACKSIZE of zero should be specified because you allocate stack for each RMX/80 task in the static task descriptors.

DEBUGGING

As mentioned in the overview of the RMX/80 Debugger, the real-time environment is a complex one in which to debug your programs. Intel provides two tools that you can use for debugging. The RMX/80 Debugger and the Intel In-Circuit Emulator (ICE). It is desirable to have both of these debugging tools at your disposal.

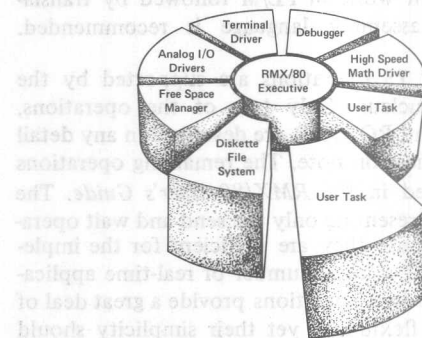


Figure 14. RMX/80 Linking

ICE enables you to use Intel Microcomputer Development System memory in place of SBC 80 memory. This allows RAM residency during your debugging as opposed to programming PROMs for each iteration. Your system may initially fail before the RMX/80 Debugger can begin operation. In this situation ICE can be used to debug your program.

APPLICATIONS

RMX/80 is suitable for a wide variety of applications. Two specific examples are presented in this application note. Each example illustrates the steps involved in using RMX/80 and provides a detailed description of the coding itself.

MINIMAL TERMINAL HANDLER

The basic functions required for a terminal handler are well defined. The handler must respond to

operator input, transmit output characters, and echo characters as they are entered. This application note describes one implementation of a minimal terminal handler.

The terminal handler presented here is not the RMX/80 Terminal Handler. It does provide some common functions and uses the same exchanges and message formats. However, many features of the RMX/80 Terminal Handler have been left out. Omitted features include special hooks to run with the Debugger, an alarm exchange, control S, Q, and O operations.

As described in the chapter on using RMX/80, the process of developing an RMX/80 application begins with the definition of the tasks and exchanges. The graphic notation is used to prepare a diagram (Figure 15) showing the tasks, exchanges, and their interaction.

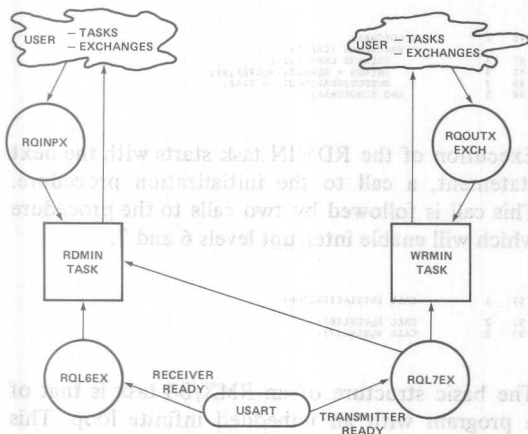


Figure 15. Minimal Terminal Handler

As shown in Figure 15, the RDMIN task waits on the RQINPX exchange for input requests. The RDMIN task also successively waits on the RQL6EX and RQL7EX exchanges. It uses the RQL6EX exchange to determine when a character has been received by the USART. The RQL7EX exchange is used to indicate when the transmitter is ready to accept another character. RDMIN uses RQL7EX for echoing input characters.

The WRMIN task waits on the RQOUTX exchange for output requests. When it receives a request, it

waits on the RQL7EX to determine when characters can be sent to the USART.

The following listing* shows the RDMIN and WRMIN tasks. These tasks provide a minimal terminal handler. The program is written in PL/M. The WRMIN task is also presented in assembly language in Appendix B. The program listing is interspersed with explanatory text. The program begins with the program segment label "MINIMAL\$TERMINAL\$HANDLER:" and a DO statement.

```
1 MINIMAL$TERMINAL$HANDLER:
DO;
```

Several macros are declared using the reserved word LITERALLY. These macros are expanded at compile time by textual substitution.

```
2 1 DECLARE TRUE LITERALLY 'OFFH';
3 1 DECLARE FOREVER LITERALLY 'WHILE TRUE';

/* SPECIAL ASCII CHARACTERS */
4 1 DECLARE
BELL LITERALLY '07H',
LF LITERALLY '0AH',
CR LITERALLY '0DH',
CONTROL$R LITERALLY '12H',
CONTROL$X LITERALLY '18H',
ESC LITERALLY '1BH',
RUBOUT LITERALLY '7FH';
```

Some macros are used to simplify the declaration of RMX/80 data structures. The structures declared here are for the exchange descriptor, interrupt exchange descriptor, and the messages used by the minimal terminal handler.

```
5 1 DECLARE EXCHANGEDESCRPTOR LITERALLY 'STRUCTURE (
MESSAGE$HEAD ADDRESS,
MESSAGE$TAIL ADDRESS,
TASK$HEAD ADDRESS,
TASK$TAIL ADDRESS,
EXCHANGE$LINK ADDRESS)';

6 1 DECLARE INT$EXCHANGEDESCRPTOR LITERALLY 'STRUCTURE (
MESSAGE$HEAD ADDRESS,
MESSAGE$TAIL ADDRESS,
TASK$HEAD ADDRESS,
TASK$TAIL ADDRESS,
EXCHANGE$LINK ADDRESS,
LINK ADDRESS,
LENGTH ADDRESS,
TYPE BYTE)';

7 1 DECLARE TH$MSG LITERALLY 'STRUCTURE (
LINK ADDRESS,
LENGTH ADDRESS,
TYPE BYTE,
HON$EXCHANGE ADDRESS,
RESPONSE$EXCHANGE ADDRESS,
STATUS ADDRESS,
BUFFER$ADDRESS ADDRESS,
COUNT ADDRESS,
ACTUAL ADDRESS)';
```

The following macros are specifically for the SBC 80/20. The macros require changes to run the minimal terminal handler on a different Single Board Computer. Intel 8253 timer/counter and 8251 USART chips are used.

*Full size listings in Appendixes A and C.

```

/* 8253 PORT ADDRESSES.
8 1 DECLARE AB253S3MODE LITERALLY '0DFH';
9 1 DECLARE AB253SCTR2 LITERALLY '0DEH';

/* 8253 COMMANDS.
10 1 DECLARE SELECTS2 LITERALLY '10000000B';
11 1 DECLARE RLSBOTH LITERALLY '00110000B';
12 1 DECLARE MODES3 LITERALLY '00000110B';
13 1 DECLARE B2400 LITERALLY '001CH';

/* 8251 PORT ADDRESSES.
14 1 DECLARE USARTIN LITERALLY '0ECH';
    USARTOUT LITERALLY '0ECH';
    USARTCONTROL LITERALLY '0E0H';

/* 8251 MODES.
15 1 DECLARE STOPS1 LITERALLY '01000000B';
16 1 DECLARE CLS LITERALLY '00001100B';
17 1 DECLARE RATES16K LITERALLY '00000110B';

/* 8251 COMMANDS.
18 1 DECLARE USARTRESET LITERALLY '01000000B';
    RTS LITERALLY '00100000B';
    ERRORSRESET LITERALLY '00010000B';
    RXE LITERALLY '00000100B';
    DTR LITERALLY '00000010B';
    TXEN LITERALLY '00000001B';

```

RDMIN and WRMIN call three RMX/80 operations. They are RQSEND, RQWAIT, and RQELVL. RQSEND and RQWAIT allow tasks to send and receive messages from exchanges. RQELVL enables a specific interrupt level.

```

19 1 RQSEND:
    PROCEDURE (EXCHANGESPOINTER,MESSAGESPOINTER) EXTERNAL;
20 2 DECLARE (EXCHANGESPOINTER,MESSAGESPOINTER) ADDRESS;
21 2 END RQSEND;

22 1 RQWAIT:
    PROCEDURE (EXCHANGESPOINTER,DELAY) ADDRESS EXTERNAL;
23 2 DECLARE (EXCHANGESPOINTER,DELAY) ADDRESS;
24 2 END RQWAIT;

25 1 RQELVL:
    PROCEDURE (LEVEL) EXTERNAL;
26 2 DECLARE LEVEL BYTE;
27 2 END RQELVL;

```

The exchange descriptors and interrupt exchange descriptors must be PUBLIC because they are referenced by the configuration module.

```

28 1 DECLARE RQINPX EXCHANGEDESSCRIPTOR PUBLIC;
29 1 DECLARE RQOUTX EXCHANGEDESSCRIPTOR PUBLIC;
30 1 DECLARE RQL6EX INTSEXCHANGEDESSCRIPTOR PUBLIC;
31 1 DECLARE RQL7EX INTSEXCHANGEDESSCRIPTOR PUBLIC;

```

The following procedure initializes the 8253 and 8251 (USART). The 8253 generates the baud rate clock (2400 baud in this example). The program sends four nulls to the USART control port to ensure that the USART is ready for a command, no matter what state it was previously in. The program then sends a reset command to the USART, followed by the mode and another command.

```

32 1 INITIALIZATION:
    PROCEDURE;
33 2 OUTPUT(AB253S3MODE) = SELECTS2 OR RLSBOTH OR MODES3;
34 2 OUTPUT(AB253SCTR2) = LOW(B2400);
35 2 OUTPUT(AB253SCTR2) = HIGH(B2400);
36 2 OUTPUT(USARTCONTROL);
    OUTPUT(USARTCONTROL);
    OUTPUT(USARTCONTROL);
    OUTPUT(USARTCONTROL) = 0;
37 2 OUTPUT(USARTCONTROL) = USARTRESET;
38 2 OUTPUT(USARTCONTROL) = STOPS1 OR CLS OR RATES16K;
39 2 OUTPUT(USARTCONTROL) = RTS OR ERRORSRESET OR
    RXE OR DTR OR TXEN;
40 2 END INITIALIZATION;

```

Tasks coded in PL/M take the form of parameterless PUBLIC procedures. The procedure declaration is followed by the variables used in RDMIN. MSGPTR receives the address of an input request message. The based-variable MSG accesses the data in the input request message. INTMSG is a dummy variable which simply receives the address of the interrupt message. BUF\$ADDRESS points to the buffer where the input characters are to be placed. The BUF array is based at the buffer pointed to by BUF\$ADDRESS.

```

41 1 RDMIN:
    PROCEDURE PUBLIC;
42 2 DECLARE (MSGPTR,INTMSG,BUF$ADDRESS) ADDRESS;
43 2 DECLARE (CHAR,PTR,I) BYTE;
44 2 DECLARE MSG BASED MSGPTR THMSG;
45 2 DECLARE (BUF BASED BUF$ADDRESS) (1) BYTE;

```

The RDMIN task echoes characters after they are read in. The ECHO\$CHAR procedure performs this function. It waits for a level 7 interrupt, indicating that the transmitter is ready for another character. ECHO\$CHAR then transmits the character.

```

46 2 ECHO$CHAR:
    PROCEDURE (CHAR);
47 3 DECLARE CHAR BYTE;
48 3 INTMSG = RQWAIT(.RQL7EX,0);
49 3 OUTPUT(USARTOUT) = CHAR;
50 3 END ECHO$CHAR;

```

Execution of the RDMIN task starts with the next statement, a call to the initialization procedure. This call is followed by two calls to the procedure which will enable interrupt levels 6 and 7.

```

51 2 CALL INITIALIZATION;
52 2 CALL RQELVL(6);
53 2 CALL RQELVL(7);

```

The basic structure of an RMX/80 task is that of a program with an imbedded infinite loop. This loop starts with the DO FOREVER statement. In the continuous loop, the task waits for an input request message. This wait is satisfied when some other task in the system sends an input request message to the RQINPX exchange. The based variable used to point to BUF is assigned from a field in the input request message, MSG.BUFFER\$ADDRESS. An index for the BUF array, PTR, and the variable CHAR are initialized.

```

54 2 DO FOREVER;
55 3 MSGPTR = RQWAIT(.RQINPX,0);
56 3 BUF$ADDRESS = MSG.BUFFER$ADDRESS - 1;
57 3 PTR = 0;
58 3 CHAR = NOT CR;

```

Task execution continues inside the next loop until a carriage return (CR) is input. An escape character (ESC) within the loop simulates a CR

which enables an exit from the loop. The task simply waits on the RQL6EX exchange for a message. This amounts to an interrupt service routine. When the wait is satisfied, the USART has received a character.

```
59 3 DO WHILE CHAR <> CR;
60 4 INTMSG = RQWAIT(.RQL6EX,0);
```

The next statement performs a whole series of operations. The character input from the USART is logically ANDed with 7FH to mask off the parity bit, assigned to the variable CHAR, and tested to determine if it is a RUBOUT character. If a RUBOUT is found, either a BELL is echoed to the terminal if there are not characters to delete in the buffer (PTR = 0), or the last character in the buffer is echoed and the pointer is decremented.

```
61 4 IF (CHAR := INPUT(USARTIN) AND 7FH) = RUBOUT THEN
62 4 DO;
63 5 IF PTR = 0 THEN
64 5 CALL ECHOSCHAR(BELL);
65 5 ELSE
66 5 DO;
67 5 CALL ECHOSCHAR(BUF(PTR));
68 5 PTR = PTR - 1;
69 5 END;
```

If CHAR is not a RUBOUT, it is tested for a CONTROL\$X. The function of a CONTROL\$X is to delete the entire line by resetting PTR to zero. After deleting the line, the system prompts the operator with a “#” character and is ready to accept a new line.

```
70 4 ELSE
71 4 DO;
72 5 IF CHAR = CONTROL$X THEN
73 5 DO;
74 5 CALL ECHOSCHAR('#');
75 5 CALL ECHOSCHAR(CR);
76 5 PTR = 0;
77 5 END;
```

The next test determines if CHAR is a CONTROL\$R. CONTROL\$R echoes the entire line that has been entered. This function is useful for displaying a line containing a number of RUBOUTs. Such lines can be difficult to interpret because RUBOUT echoes deleted characters. Because CONTROL\$R echoes only the remaining data in the buffer, it eliminates “garbage” from the display.

```
78 5 ELSE
79 5 DO;
80 5 IF CHAR = CONTROL$R THEN
81 5 DO;
82 5 CALL ECHOSCHAR(CR);
83 5 DO I = 1 TO PTR;
84 5 CALL ECHOSCHAR(BUF(I));
85 5 END;
86 5 END;
```

The character is then placed in the buffer unless the end of the buffer has been reached. If the buffer is full, a BELL is sent to the terminal.

```
87 6 ELSE
88 6 DO;
89 6 IF PTR < MSG.COUNT THEN
90 6 BUF(PTR := PTR+1) = CHAR;
91 6 ELSE
92 6 IF CHAR <> CR THEN
93 6 CHAR = BELL;
```

The last test is for an ESC character. It is echoed as a “\$” and is treated as if a CR were entered.

```
94 7 IF CHAR = ESC THEN
95 7 DO;
96 7 CALL ECHOSCHAR('$');
97 7 CHAR = CR;
98 7 END;
99 7 CALL ECHOSCHAR(CHAR);
100 7 END;
101 6 END;
102 5 END;
103 4 END;
```

The program places a line feed (LF) at the end of the buffer when an exit is forced by a CR or an ESC. The input request message actual character count (MSG.ACTUAL) and the status (MSG.STATUS) are set before sending the message to its response exchange.

```
104 3 IF PTR < MSG.COUNT THEN
105 3 BUF(PTR:=PTR+1) = LF;
106 3 MSG.ACTUAL = PTR;
107 3 MSG.STATUS = 0;
108 3 CALL RQEND(MSG,RESPONSESEXCHANGE,MSGPTR);
109 3 CALL ECHOSCHAR(LF);
110 3 END;
111 2 END RQSNIN;
```

The WRMIN task begins by enabling interrupt level 7. Note that no other initialization is performed before WRMIN waits for an output request message to arrive at the RQOUTX exchange. Here correct operation depends on the fact that RDMIN has a higher priority than WRMIN. Were this not the case, WRMIN could try to transmit a message before the 8253 and 8251 have been set up.

```
112 1 WRMIN:
113 2 PROCEDURE PUBLIC;
114 2 DECLARE (MSGPTR,INTMSG,BUF$ADDRESS) ADDRESS;
115 2 DECLARE PTR BYTE;
116 2 DECLARE MSG BASED MSGPTR THMSG;
117 2 DECLARE (BUF BASED BUF$ADDRESS) (1) BYTE;
118 2 CALL RQELVL(7);
119 2 DO FOREVER;
120 2 MSGPTR = RQWAIT(.RQOUTX,0);
121 2 BUF$ADDRESS = MSG.BUF$ADDRESS - 1;
```

The next loop transmits all of the characters specified by the output request message. Once again, the interrupt service routine is implemented by simply waiting on the RQL7EX exchange for a transmitter ready interrupt message. When this message is received, the next character in the buffer is transmitted.


```

121 3      DO PTR = 1 TO MSG.COUNT;
122 4      INTMSG = RQWAIT(.RQL7EX,8);
123 4      OUTPUT(USARTSOUT) = BUF(PTR);
124 4      END;

```

The WRMIN task concludes by setting the actual count and status, and then sends the output request message to its response exchange.

```

125 3      MSG.ACTUAL = MSG.COUNT;
126 3      MSG.STATUS = #1;
127 3      CALL RQSEND(MSG,RESPONSE$EXCHANGE,MSGPTR);
128 3      END;
129 2      END WRMIN;

```

Using the macros provided on the RMX/80 diskette, the following static task descriptors (STD) should be placed in your configuration module.

```

STD      RDMIN,64,112,0
STD      WRMIN,64,128,0

```

The entries in the STD are interpreted as follows.

STD NAME,STKLEN,PRI,EXCH
where:

- NAME = the symbolic name assigned to the task associated with the STD
- STKLEN = the number of bytes allocated to the task stack
- PRI = the task priority level
- EXCH = an optional field, usually 0

Priorities of 112 and 128 have been assigned to RDMIN and WRMIN because they correspond to hardware interrupt levels 6 and 7.

The following exchange addresses should be placed in your configuration module.

```

XCHADR      RQINPX
XCHADR      RQOUTX
XCHADR      RQL6EX
XCHADR      RQL7EX

```

The XCHADR macro only requires the address of the exchange descriptor.

Characters typed at the terminal are ignored unless an input request message has been received. Thus, type-ahead is not a built-in feature. However, if type-ahead is desired, it is sufficient to ensure that input requests are always queued for the RDMIN task and that the full input buffers are sent to an exchange that queues full buffers.

This can easily be accomplished by sending several input requests to the RQINPX. These input requests have the address of a "full-buffer" exchange as the response exchange and the RQINPX exchange as the home exchange. Then, tasks needing terminal input wait on the "full-buffer" exchange and send the message to the home exchange when finished.

CLOSED-LOOP ANALOG CONTROL

In the next example, a closed-loop analog control subsystem using the Intel SBC 711 analog-to-digital board illustrates task scheduling and synchronization in a process control application. In general, the subsystem samples an analog input at specified intervals, converts the data to temperature in degrees centigrade, and then — based upon programmed temperature limits — controls a heating element. The algorithm used provides a 2-position controller with neutral intermediate zone (or simply "bang-bang" control). The control algorithm is shown in Figure 16.

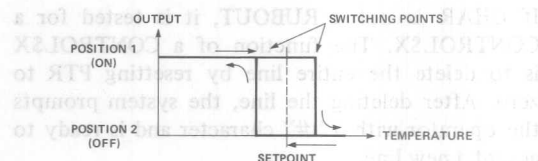


Figure 16. 2-Position Controller with Neutral Intermediate Zone

The graphic notation in Figure 17 diagrams the tasks, exchanges, and their interaction.

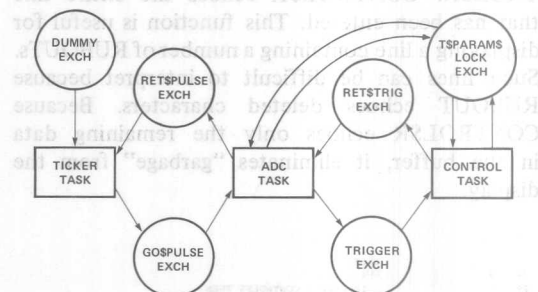


Figure 17. Analog Subsystem

This application includes three tasks and six associated exchanges. The TICKER task schedules the ADC task. TICKER has a very high priority because nothing else in the system should interfere with its scheduling activities. It is also a very short task since it repetitively executes a timed wait and then handshakes a message.

TICKER schedules the ADC task. The ADC task services the A/D converter. After obtaining data from the A/D it handshakes with the CONTROL task to signal that data is ready for processing. The ADC task is assigned a priority equivalent to the level of the hardware interrupt from the A/D. Clearly, calculations should not be performed at that priority.

Thus, CONTROL performs the processing function at a lower priority. The CONTROL task used the T\$PARAM\$LOCK exchange to govern access to the control parameters. This avoids problems resulting when some other task is updating the parameters at the same time the CONTROL task is using them for testing.

As in the minimal terminal handler, the following listing contains the complete analog subsystem tasks and is interspersed with explanatory text. The program begins with the program segment label "ATOD:" and a DO statement.

```

1      ATOD:
      DO:

      $INCLUDE:(F1:COMMON.ELT)
      DECLARE TRUE LITERALLY 'OFFH';
      DECLARE FALSE LITERALLY 'ODH';
      DECLARE BOOLEAN LITERALLY 'BYTE';
      DECLARE FOREVER LITERALLY 'WHILE';

      $INCLUDE:(F1:EXCH.ELT)
      DECLARE EXCHANGES$DESCRIPTOR LITERALLY 'STRUCTURE (
      = MESSAGE$HEAD ADDRESS,
      = MESSAGE$TAIL ADDRESS,
      = TASK$HEAD ADDRESS,
      = TASK$TAIL ADDRESS,
      = EXCHANGES$LINK ADDRESS)';

      $INCLUDE:(F1:IED.ELT)
      DECLARE INT$EXCHANGES$DESCRIPTOR LITERALLY 'STRUCTURE (
      = MESSAGE$HEAD ADDRESS,
      = MESSAGE$TAIL ADDRESS,
      = TASK$HEAD ADDRESS,
      = TASK$TAIL ADDRESS,
      = EXCHANGES$LINK ADDRESS,
      = LINK ADDRESS,
      = LENGTH ADDRESS,
      = TYPE BYTE)';

      $INCLUDE:(F1:MSG.ELT)
      DECLARE MSG$HDR LITERALLY '
      = LINK ADDRESS,
      = LENGTH ADDRESS,
      = TYPE BYTE,
      = ROME$EXCHANGE ADDRESS,
      = RESPONSE$EXCHANGE ADDRESS';

      $INCLUDE:(F1:MSD.ELT)
      DECLARE MSD$DESCRIPTOR LITERALLY 'STRUCTURE (
      = MSG$HDR,
      = REMAINDER() BYTE)';

      $INCLUDE:(F1:INTRPT.EXT)
      ROENDI:
      = PROCEDURE EXTERNAL;

```

```

11 2 = END ROENDI;
12 1 = RQELVL:
13 2 = PROCEDURE (LEVEL) EXTERNAL;
14 2 = DECLARE LEVEL BYTE;
15 2 = END RQELVL;
16 1 = RQDLVL:
17 2 = PROCEDURE (LEVEL) EXTERNAL;
18 2 = DECLARE LEVEL BYTE;
19 2 = END RQDLVL;
20 1 = RQSETV:
21 2 = PROCEDURE (PROC,LEVEL) EXTERNAL;
22 2 = DECLARE PROC ADDRESS;
23 2 = DECLARE LEVEL BYTE;
24 2 = END RQSETV;

25 1 = $INCLUDE:(F1:SYNCH.EXT)
26 1 = RQSEND:
27 2 = PROCEDURE (EXCHANGES$POINTER,MESSAGES$POINTER) EXTERNAL;
28 2 = DECLARE (EXCHANGES$POINTER,MESSAGES$POINTER) ADDRESS;
29 2 = END RQSEND;

30 1 = RQWAIT:
31 2 = PROCEDURE (EXCHANGES$POINTER,DELAY) ADDRESS EXTERNAL;
32 2 = DECLARE (EXCHANGES$POINTER,DELAY) ADDRESS;
33 2 = END RQWAIT;

34 1 = RQACPT:
35 2 = PROCEDURE (EXCHANGES$POINTER) ADDRESS EXTERNAL;
36 2 = DECLARE EXCHANGES$POINTER ADDRESS;
37 2 = END RQACPT;

38 1 = RQISND:
39 2 = PROCEDURE (IED$PTR) EXTERNAL;
40 2 = DECLARE IED$PTR ADDRESS;
41 2 = END RQISND;

```

Additional macros are declared to aid in the use of the SBC 711 analog-to-digital board.

```

/* SBC 711 ANALOG TO DIGITAL BOARD */
34 1 DECLARE ADC$BASE ADDRESS AT (OPTION);
35 1 DECLARE COMMAND$REGISTER BYTE AT (.ADC$BASE+0);
36 1 DECLARE STATUS$REGISTER BYTE AT (.ADC$BASE+1);
37 1 DECLARE FIRST$CHANNEL$REGISTER BYTE AT (.ADC$BASE+2);
38 1 DECLARE LAST$CHANNEL$REGISTER BYTE AT (.ADC$BASE+3);
39 1 DECLARE CLEAR$INTERRUPT$REQUEST BYTE AT (.ADC$BASE+4);
40 1 DECLARE ADC$DATA$REGISTER ADDRESS AT (.ADC$BASE+5);

41 1 DECLARE GOSBIT LITERALLY '1';
42 1 DECLARE AUTO$INCREMENT$ENABLE LITERALLY '2';
43 1 DECLARE BUSY LITERALLY '8';
44 1 DECLARE EOS$INTERRUPT$ENABLE LITERALLY '10H';
45 1 DECLARE EOC$INTERRUPT$ENABLE LITERALLY '20H';
46 1 DECLARE END$OF$SCAN LITERALLY '40H';
47 1 DECLARE END$OF$CONVERSION LITERALLY '80H';

```

The exchange descriptors and the interrupt exchange descriptors are declared.

```

48 1 DECLARE DUMMY EXCHANGES$DESCRIPTOR PUBLIC;
49 1 DECLARE RET$PULSE EXCHANGES$DESCRIPTOR PUBLIC;
50 1 DECLARE GOS$PULSE EXCHANGES$DESCRIPTOR PUBLIC;
51 1 DECLARE TRIGGER EXCHANGES$DESCRIPTOR PUBLIC;
52 1 DECLARE RET$TRIG EXCHANGES$DESCRIPTOR PUBLIC;
53 1 DECLARE RQ2$EX INT$EXCHANGES$DESCRIPTOR;

```

The CONTROL task uses an external data structure to obtain operating parameters. This data structure (BOX\$PARAMS) has an exchange associated with it (T\$PARAM\$LOCK) that is used to provide mutual exclusion, ensuring that only one task accesses the data structure at a time.

```

54 1 DECLARE T$PARAM$LOCK EXCHANGES$DESCRIPTOR EXTERNAL;

55 1 DECLARE BOX$PARAMS(5) STRUCTURE(
      = SET$POINT ADDRESS,
      = ERROR ADDRESS,
      = OFFSET ADDRESS,
      = SAMPLES ADDRESS,
      = COUNT ADDRESS,
      = ACCUM ADDRESS,
      = READING ADDRESS ) EXTERNAL;

```

TICKER, the scheduler task, has an initialization sequence in which it sets up two messages and sends them to the RET\$PULSE exchange. Then it enters an infinite loop where it waits on the DUMMY exchange for 250 milliseconds. After the timed wait is complete, TICKER passes a message from the RET\$PULSE exchange to the GO\$PULSE exchange. In effect this is a handshake, checking to see that the ADC task has completed its last operation and then signaling it to perform another.

```

56 1  TICKERTASK:
57 2  PROCEDURE PUBLIC;
58 2  DECLARE MSG ADDRESS;
59 2  DECLARE PULSMSG(2) STRUCTURE (
60 2  MSGHDR );
61 2  PULSMSG(0).LENGTH = SIZE(PULSMSG(0));
62 2  PULSMSG(0).TYPE = 65;
63 2  CALL RQSEND(.RET$PULSE, PULSMSG(0));
64 2  CALL RQSEND(.RET$PULSE, PULSMSG(1));
65 2  DO FOREVER;
66 2  MSG = RQWAIT(.DUMMY, 5);
67 2  MSG = RQWAIT(.RET$PULSE, 0);
68 2  CALL RQSEND(.GO$PULSE, MSG);
69 2  END;

```

Scheduled by TICKER, the ADC task performs the A/D sampling. It begins by setting up TRIGGERSMSG and enabling the level 2 interrupt from the A/D. Inside the ADC task continuous loop, messages are passed from the GO\$PULSE exchange to the RET\$PULSE exchange. Then it waits for access to the BOX\$PARAMS data structure. When the ADC task has access, it loops through the A/D channels, accumulating readings in BOX\$PARAMS. After all the A/D channels are sampled and the BOX\$PARAMS readings updated, the LOCK\$MSG is returned to the TSPARAM\$LOCK exchange. The ADC task concludes the continuous loop by handshaking a message with the CONTROL task.

```

69 1  ADCTASK:
70 2  PROCEDURE PUBLIC;
71 2  DECLARE TRIGGERSMSG STRUCTURE (
72 2  MSGHDR );
73 2  DECLARE (TMSG, MSG, LOCK$MSG) ADDRESS;
74 2  DECLARE I BYTE;
75 2  DECLARE GAIN LITERALLY '00';
76 2  DECLARE NSCHNLS LITERALLY '5';
77 2  TRIGGERSMSG.LENGTH = SIZE(TRIGGERSMSG);
78 2  TRIGGERSMSG.TYPE = 65;
79 2  CALL RQSEND(.RET$TRIG, TRIGGERSMSG);
80 2  DO FOREVER;
81 2  MSG = RQWAIT(.GO$PULSE, 0);
82 2  CALL RQSEND(.RET$PULSE, MSG);
83 2  LOCK$MSG = RQWAIT(.TSPARAM$LOCK, 0);
84 2  DO I = 0 TO NSCHNLS-1;
85 2  FIRSTCHANNEL$REGISTER = BOX$PARAMS(I).CHANNEL
86 2  + ROL(GAIN, 6);
87 2  MSG = RQWAIT(.ROLPEX, 0);
88 2  COMMAND$REGISTER = 0;
89 2  BOX$PARAMS(I).ACCUM = BOX$PARAMS(I).ACCUM
90 2  + ADCDATA$REGISTER;
91 2  END;
92 2  CALL RQSEND(.TSPARAM$LOCK, LOCK$MSG);
93 2  TMSG = RQWAIT(.RET$TRIG, 0);
94 2  CALL RQSEND(.TRIGGER, TMSG);
95 2  END;

```

The CONTROL task waits for a message from the ADC task signaling that A/D readings have been taken and are ready for further processing. It completes the handshake by sending the message to the RET\$TRIG exchange. Then, as in the ADC task, accesses the BOX\$PARAMS data structure.

Inside the next loop, the readings are averaged, scaled, offset, and tested. Appropriate action is taken to turn the heating elements on or off. The loop concludes by returning the message to the TSPARAM\$LOCK exchange.

```

95 1  CONTROLTASK:
96 2  PROCEDURE PUBLIC;
97 2  DECLARE (LOCK$MSG, TMSG) ADDRESS;
98 2  DECLARE I BYTE;
99 2  DECLARE NSCHNLS LITERALLY '5';
100 2  DECLARE TURN$LAMP$ON
101 2  LITERALLY 'OUTPUT(OETH)=SHL(I,1)';
102 2  DECLARE TURN$LAMP$OFF
103 2  LITERALLY 'OUTPUT(OETH)=SHL(I,1)+1';
104 2  DECLARE SETUP$8255 LITERALLY 'OUTPUT(OETH)=80H';
105 2  SETUP$8255 = OUTPUT(OETH) = 0FFH;
106 2  DO FOREVER;
107 2  MSG = RQWAIT(.TRIGGER, 0);
108 2  CALL RQSEND(.RET$TRIG, MSG);
109 2  LOCK$MSG = RQWAIT(.TSPARAM$LOCK, 0);
110 2  DO I = 0 TO NSCHNLS-1;
111 2  BOX$PARAMS(I).COUNT = BOX$PARAMS(I).COUNT + 1;
112 2  IF BOX$PARAMS(I).COUNT
113 2  = BOX$PARAMS(I).SAMPLES THEN
114 2  DO;
115 2  BOX$PARAMS(I).READING
116 2  = (BOX$PARAMS(I).ACCUM
117 2  / BOX$PARAMS(I).SAMPLES) / 38
118 2  + BOX$PARAMS(I).OFFSET;
119 2  IF T <= BOX$PARAMS(I).SET$POINT
120 2  - BOX$PARAMS(I).ERROR THEN
121 2  TURN$LAMP$ON;
122 2  ELSE IF T >= BOX$PARAMS(I).SET$POINT
123 2  + BOX$PARAMS(I).ERROR THEN
124 2  TURN$LAMP$OFF;
125 2  BOX$PARAMS(I).ACCUM = BOX$PARAMS(I).COUNT = 0;
126 2  END;
127 2  END;
128 2  CALL RQSEND(.TSPARAM$LOCK, LOCK$MSG);
129 2  END;
130 2  END CONTROLTASK;
131 1  END ATOD;

```

SUMMARY/CONCLUSIONS

The purpose of this application note is to introduce you to the Intel RMX/80, Real-Time Multi-tasking Executive. The general framework of RMX/80 was discussed, including the nucleus and extensions.

This application note described the steps involved in using RMX/80. Key emphasis has been placed on the need to fully define the tasks and exchanges in your application using graphic notation.

Applications have been presented to demonstrate task communication, synchronization, and mutual exclusion in a minimal terminal handler and an analog subsystem. The tasks responded to real-time asynchronous events such as USART and A/D interrupts.

RMX/80 represents a significant step in the sophistication of microcomputer software. Its ease of use, flexibility, and power should enable you to quickly implement real-time software for your applications.

APPENDIX A

MINITH PL/M LISTING

```

1      MINIMAL$TERMINAL$HANDLER:

      DO;

2      1      DECLARE TRUE LITERALLY '0FFH';
3      1      DECLARE FOREVER LITERALLY 'WHILE TRUE';

      /* SPECIAL ASCII CHARACTERS */
4      1      DECLARE
          BELL          LITERALLY '07H',
          LF            LITERALLY '0AH',
          CR            LITERALLY '0DH',
          CONTROL$R     LITERALLY '12H',
          CONTROL$X     LITERALLY '18H',
          ESC           LITERALLY '1BH',
          RUBOUT        LITERALLY '7FH';

5      1      DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
          MESSAGE$HEAD ADDRESS,
          MESSAGE$TAIL ADDRESS,
          TASK$HEAD ADDRESS,
          TASK$TAIL ADDRESS,
          EXCHANGE$LINK ADDRESS);'

6      1      DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
          MESSAGE$HEAD ADDRESS,
          MESSAGE$TAIL ADDRESS,
          TASK$HEAD ADDRESS,
          TASK$TAIL ADDRESS,
          EXCHANGE$LINK ADDRESS,
          LINK ADDRESS,
          LENGTH ADDRESS,
          TYPE BYTE);'

7      1      DECLARE TH$MSG LITERALLY 'STRUCTURE(
          LINK ADDRESS,
          LENGTH ADDRESS,
          TYPE BYTE,
          HOME$EXCHANGE ADDRESS,
          RESPONSE$EXCHANGE ADDRESS,
          STATUS ADDRESS,
          BUFFER$ADDRESS ADDRESS,
          COUNT ADDRESS,
          ACTUAL ADDRESS);'

      /*
      8253 PORT ADDRESSES.
      */
8      1      DECLARE A8253$MODE LITERALLY '0DFH';
9      1      DECLARE A8253$CTR2 LITERALLY '0DEH';

```

```

/*
8253 COMMANDS.
*/
10 1 DECLARE SELECT$2 LITERALLY '10000000B';
11 1 DECLARE RL$BOTH LITERALLY '00110000B';
12 1 DECLARE MODE$3 LITERALLY '00000110B';
13 1 DECLARE B2400 LITERALLY '001CH';

/*
8251 PORT ADDRESSES.
*/
14 1 DECLARE USART$IN LITERALLY '0ECH',
        USART$OUT LITERALLY '0ECH',
        USART$CONTROL LITERALLY '0EDH';

/*
8251 MODES.
*/
15 1 DECLARE STOP$1 LITERALLY '01000000B';
16 1 DECLARE CL8 LITERALLY '00001100B';
17 1 DECLARE RATE$16X LITERALLY '00000010B';

/*
8251 COMMANDS.
*/
18 1 DECLARE USART$RESET LITERALLY '01000000B',
        RTS LITERALLY '00100000B',
        ERROR$RESET LITERALLY '00010000B',
        RXE LITERALLY '00000100B',
        DTR LITERALLY '00000010B',
        TXEN LITERALLY '00000001B';

19 1 RQSEND:
        PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
20 2 DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
21 2 END RQSEND;

22 1 RQWAIT:
        PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
23 2 DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
24 2 END RQWAIT;

25 1 RQELVL:
        PROCEDURE (LEVEL) EXTERNAL;
26 2 DECLARE LEVEL BYTE;
27 2 END RQELVL;

28 1 DECLARE RQINPX EXCHANGE$DESCRIPTOR PUBLIC;
29 1 DECLARE RQOUTX EXCHANGE$DESCRIPTOR PUBLIC;

30 1 DECLARE RQL6EX INT$EXCHANGE$DESCRIPTOR PUBLIC;
31 1 DECLARE RQL7EX INT$EXCHANGE$DESCRIPTOR PUBLIC;

32 1 INITIALIZATION:
        PROCEDURE;
33 2 OUTPUT(A8253$MODE) = SELECT$2 OR RL$BOTH OR MODE$3;
34 2 OUTPUT(A8253$CTR2) = LOW(B2400);

```


35	2	OUTPUT(A8253\$CTR2) = HIGH(B2400);	3	02
36	2	OUTPUT(USART\$CONTROL),	3	03
		OUTPUT(USART\$CONTROL),	7	18
		OUTPUT(USART\$CONTROL),	7	23
		OUTPUT(USART\$CONTROL) = 0;	7	28
37	2	OUTPUT(USART\$CONTROL) = USART\$RESET;	8	33
38	2	OUTPUT(USART\$CONTROL) = STOP\$1 OR CL8 OR RATE\$16X;	8	38
39	2	OUTPUT(USART\$CONTROL) = RTS OR ERROR\$RESET OR	8	43
		RXE OR DTR OR TXEN;	8	48
40	2	END INITIALIZATION;	8	53
41	1	RD\$MIN: = PTR+1;	7	58
		PROCEDURE PUBLIC;	7	63
42	2	DECLARE (MSGPTR,INTMSG,BUF\$ADDRESS) ADDRESS;	8	68
43	2	DECLARE (CHAR,PTR,I) BYTE;	8	73
44	2	DECLARE MSG BASED MSGPTR TH\$MSG;	8	78
45	2	DECLARE (BUF BASED BUF\$ADDRESS) (1) BYTE;	8	83
46	2	ECHO\$CHAR:	7	88
		PROCEDURE (CHAR);	7	93
47	3	DECLARE CHAR BYTE;	8	98
48	3	INTMSG = RQWAIT(.RQL7EX,0);	8	103
49	3	OUTPUT(USART\$OUT) = CHAR;	7	108
50	3	END ECHO\$CHAR;	7	113
51	2	CALL INITIALIZATION;	8	118
52	2	CALL RQELVL(6);	4	123
53	2	CALL RQELVL(7);	3	128
54	2	DO FOREVER;	3	133
55	3	MSGPTR = RQWAIT(.RQINPX,0);	3	138
56	3	BUF\$ADDRESS = MSG.BUFFER\$ADDRESS - 1;	3	143
57	3	PTR = 0;	3	148
58	3	CHAR = NOT CR;	3	153
59	3	DO WHILE CHAR <> CR;	3	158
60	4	INTMSG = RQWAIT(.RQL6EX,0);	1	163
61	4	IF (CHAR := INPUT(USART\$IN) AND 7FH) = RUBOUT THEN	1	168
62	4	DO;	1	173
63	5	IF PTR = 0 THEN	2	178
64	5	CALL ECHO\$CHAR(BELL);	2	183
65	5	DO;	2	188
66	6	CALL ECHO\$CHAR(BUF(PTR));	2	193
67	6	PTR = PTR - 1;	2	198
68	6	END;	2	203
69	5	END;	2	208
70	4	DO;	3	213
71	5	IF CHAR = CONTROL\$X THEN	4	218
72	5	DO;	4	223
73	6	CALL ECHO\$CHAR('#');	4	228
74	6	CALL ECHO\$CHAR(CR);	4	233
75	6	CALL ECHO\$CHAR(LF);	4	238
76	6	PTR = 0;	4	243
77	6	END;	4	248
78	5	ELSE	4	253
		DO;	4	258

79	6	IF CHAR = CONTROL\$K THEN	2
80	6	DO;	2
81	7	CALL ECHO\$CHAR(CR);	
82	7	CALL ECHO\$CHAR(LF);	
83	7	DO I = 1 TO PTR;	
84	8	CALL ECHO\$CHAR(BUF(I));	
85	8	END;	
86	7	ELSE	
87	6	DO;	
88	7	IF PTR < MSG.COUNT THEN	
89	7	BUF(PTR := PTR+1) = CHAR;	
		ELSE	
90	7	DO;	
91	8	IF CHAR <> CR THEN	
92	8	CHAR = BELL;	
93	8	END;	
94	7	IF CHAR = ESC THEN	
95	7	DO;	
96	8	CALL ECHO\$CHAR('\$');	
97	8	CHAR = CR;	
98	8	END;	
99	7	CALL ECHO\$CHAR(CHAR);	
100	7	END;	
101	6	END;	
102	5	END;	
103	4	END;	
104	3	IF PTR < MSG.COUNT THEN	
105	3	BUF(PTR:=PTR+1) = LF;	
106	3	MSG.ACTUAL = PTR;	
107	3	MSG.STATUS = 0;	
108	3	CALL RQSEND(MSG.RESPONSE\$EXCHANGE,MSGPTR);	
109	3	CALL ECHO\$CHAR(LF);	
110	3	END;	
111	2	END RD\$MIN;	
112	1	WR\$MIN: 0, X\$EXCH, 0);	
113	2	PROCEDURE PUBLIC;	
114	2	DECLARE (MSGPTR,INTMSG,BUF\$ADDRESS) ADDRESS;	
115	2	DECLARE PTR BYTE;	
116	2	DECLARE MSG BASED MSGPTR TH\$MSG;	
		DECLARE (BUF BASED BUF\$ADDRESS) (1) BYTE;	
117	2	CALL RQELVL(7);	
118	2	DO FOREVER;	
119	3	MSGPTR = RQWAIT(.RQOUTX,0);	
120	3	BUF\$ADDRESS = MSG.BUFFER\$ADDRESS - 1;	
121	3	DO PTR = 1 TO MSG.COUNT;	
122	4	INTMSG = RQWAIT(.RQL7EX,0);	
123	4	OUTPUT(USART\$OUT) = BUF(PTR);	
124	4	END;	
125	3	MSG.ACTUAL = MSG.COUNT;	
126	3	MSG.STATUS = 0;	
127	3	CALL RQSEND(MSG.RESPONSE\$EXCHANGE,MSGPTR);	
128	3	END;	
129	2	END WR\$MIN;	
130	1	END MINIMAL\$TERMINAL\$HANDLER;	

APPENDIX B

WRMIN ASSEMBLY LANGUAGE LISTING

LOC	OBJ	SEQ	SOURCE STATEMENT
		1	NAME WRMIN
		2	EXTRN RQELVL,RQOUTX,RQWAIT,RQSEND
		3	PUBLIC WRMIN,RQL7EX
00EC		4	DATOUT EQU OECH ; USART OUTPUT PORT ADR
		5	CSEG
		6	WRMIN:
0000	0E07	7	MVI C,7
0002	CD0000	8	CALL RQELVL ; ENABLE INTERRUPT LVL 7
		9	WRO:
0005	110000	10	LXI D,0
0008	010000	11	LXI B,RQOUTX
000B	CD0000	12	CALL RQWAIT ; WAIT FOR OUTPUT RQST
000E	E5	13	PUSH H ; PUSH MESSAGE ADDRESS
000F	110700	14	LXI D,7
0012	19	15	DAD D
0013	4E	16	MOV C,M ; GET RESPONSE EXCHANGE
0014	23	17	INX H
0015	46	18	MOV B,M
0016	23	19	INX H
0017	C5	20	PUSH B ; PUSH RESPONSE EXCHANGE
0018	3600	21	MVI M,0 ; STATUS = 0
001A	23	22	INX H
001B	3600	23	MVI M,0
001D	23	24	INX H
001E	5E	25	MOV E,M ; GET BUFFER ADR IN DE
001F	23	26	INX H
0020	56	27	MOV D,M
0021	23	28	INX H
0022	4E	29	MOV C,M ; GET COUNT IN BC
0023	23	30	INX H
0024	46	31	MOV B,M
0025	23	32	INX H
0026	71	33	MOV M,C ; ACTUAL = COUNT
0027	23	34	INX H
0028	70	35	MOV M,B
		36	WR1:
0029	78	37	MOV A,B
002A	B1	38	ORA C
002B	CA4300	39	JZ WR2 ; EXIT LOOP IF COUNT = 0
002E	C5	40	PUSH B
002F	D5	41	PUSH D
0030	110000	42	LXI D,0
0033	010000	43	LXI B,RQL7EX
0036	CD0000	44	CALL RQWAIT ; WAIT FOR TXRDY INTRPT
0039	D1	45	POP D
003A	C1	46	POP B
003B	1A	47	LDAX D
003C	13	48	INX D
003D	D3EC	49	OUT DATOUT ; TRANSMIT NEXT CHAR
003F	0B	50	DCX B
0040	C32900	51	JMP WR1
		52	WR2:

0043 C1 53 POP B ; BC = RESPONSE EXCHANGE
 0044 D1 54 POP D ; DE = MSG ADDRESS
 0045 CD0000 E 55 CALL RQSEND ; SEND MSG TO RESP. EXCH
 0048 C30500 C 56 JMP WRO

57 ;
 58 DSEG
 59 RQL7EX:
 000F 60 DS 15
 61 ;
 62 END

0000 0507 MVI C, 7
 0002 00000 CALL ROEVL ; ENABLE INTERRUPT LVL 7
 0005 110000 LXI D, 0
 0008 010000 LXI B, RQOUTX
 000B 000000 CALL RQWAIT ; WAIT FOR OUTPUT RQST
 000E 05 PUSH H ; PUSH MESSAGE ADDRESS
 000F 110700 LXI D, 7
 0012 10 DAD D
 0015 10 MOV C, M ; GET RESPONSE EXCHANGE
 0018 10 INX H
 001B 10 MOV B, M
 001E 10 INX H
 0021 10 PUSH B ; PUSH RESPONSE EXCHANGE
 0024 10 MVI M, 0 ; STATUS = 0
 0027 10 INX H
 002A 10 MVI M, 0
 002D 10 INX H
 0030 10 MOV E, M ; GET BUFFER ADDR IN DE
 0033 10 INX H
 0036 10 MOV D, M
 0039 10 INX H
 003C 10 MOV C, M ; GET COUNT IN BC
 003F 10 INX H
 0042 10 MOV B, M
 0045 10 INX H
 0048 10 MOV M, C ; ACTUAL = COUNT
 004B 10 INX H
 004E 10 MOV M, B
 0051 10 MOV A, B
 0054 10 ORA C
 0057 10 JZ WRS ; EXIT LOOP IF COUNT = 0
 005A 10 PUSH B
 005D 10 PUSH D
 0060 10 LXI D, 0
 0063 10 LXI B, RQL7EX
 0066 10 CALL RQWAIT ; WAIT FOR TXRDY INTPT
 0069 10 POP B
 006C 10 POP D
 006F 10 LDAX D
 0072 10 INX D
 0075 10 OUT DATOUT ; TRANSMIT NEXT CHAR
 0078 10 DCX B
 007B 10 JMP WRS

APPENDIX C

ATOD PL/M LISTING

```

1      ATOD:
      DO;

      $INCLUDE(:F1:COMMON.ELT)

2      1 = DECLARE TRUE LITERALLY 'OFFH';
3      1 = DECLARE FALSE LITERALLY 'OOH';
4      1 = DECLARE BOOLEAN LITERALLY 'BYTE';
5      1 = DECLARE FOREVER LITERALLY 'WHILE';

      $INCLUDE(:F1:EXCH.ELT)

6      1 = DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      MESSAGE$HEAD ADDRESS,
      MESSAGE$TAIL ADDRESS,
      TASK$HEAD ADDRESS,
      TASK$TAIL ADDRESS,
      EXCHANGE$LINK ADDRESS)';

      $INCLUDE(:F1:IED.ELT)

7      1 = DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      MESSAGE$HEAD ADDRESS,
      MESSAGE$TAIL ADDRESS,
      TASK$HEAD ADDRESS,
      TASK$TAIL ADDRESS,
      EXCHANGE$LINK ADDRESS,
      LINK ADDRESS,
      LENGTH ADDRESS,
      TYPE BYTE)';

      $INCLUDE(:F1:MSG.ELT)

8      1 = DECLARE MSG$HDR LITERALLY '
      LINK ADDRESS,
      LENGTH ADDRESS,
      TYPE BYTE,
      HOME$EXCHANGE ADDRESS,
      RESPONSE$EXCHANGE ADDRESS';

9      1 = DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
      MSG$HDR,
      REMAINDER(1) (BYTE)';

10     1 = $INCLUDE(:F1:INTRPT.EXT)
11     2 = RQENDI;

12     1 = RQELVL;
13     2 = $INCLUDE(:F1:INTRPT.EXT)
14     2 = $INCLUDE(:F1:INTRPT.EXT)

```



```

47 1 DECLARE END$OF$CONVERSION LITERALLY '80H';
48 1 DECLARE DUMMY EXCHANGE$DESCRIPTOR PUBLIC;
49 1 DECLARE RET$PULSE EXCHANGE$DESCRIPTOR PUBLIC;
50 1 DECLARE GO$PULSE EXCHANGE$DESCRIPTOR PUBLIC;
51 1 DECLARE TRIGGER EXCHANGE$DESCRIPTOR PUBLIC;
52 1 DECLARE RET$TRIG EXCHANGE$DESCRIPTOR PUBLIC;

53 1 DECLARE RQL2EX INT$EXCHANGE$DESCRIPTOR;

54 1 DECLARE T$PARAM$LOCK EXCHANGE$DESCRIPTOR EXTERNAL;

55 1 DECLARE BOX$PARAMS(5) STRUCTURE(
    CHANNEL BYTE,
    SET$POINT ADDRESS,
    ERROR ADDRESS,
    OFFSET ADDRESS,
    SAMPLES ADDRESS,
    COUNT ADDRESS,
    ACCUM ADDRESS,
    READING ADDRESS ) EXTERNAL;

56 1 TICKER$TASK:
    PROCEDURE PUBLIC;
57 2 DECLARE MSG ADDRESS;
58 2 DECLARE PULSE$MSG(2) STRUCTURE (
    MSG$HDR );

59 2 PULSE$MSG(0).LENGTH,
    PULSE$MSG(1).LENGTH = SIZE(PULSE$MSG(0));
60 2 PULSE$MSG(0).TYPE,
    PULSE$MSG(1).TYPE = 65;
61 2 CALL RQSEND(.RET$PULSE,.PULSE$MSG(0));
62 2 CALL RQSEND(.RET$PULSE,.PULSE$MSG(1));

63 2 DO FOREVER;
64 3 MSG = RQWAIT(.DUMMY,5);
65 3 MSG = RQWAIT(.RET$PULSE,0);
66 3 CALL RQSEND(.GO$PULSE,MSG);
67 3 END;

68 2 END TICKER$TASK;

69 1 ADC$TASK:
    PROCEDURE PUBLIC;
70 2 DECLARE TRIGGER$MSG STRUCTURE (
    MSG$HDR );
71 2 DECLARE (T$MSG,MSG,LOCK$MSG) ADDRESS;
72 2 DECLARE I BYTE;
73 2 DECLARE GAIN LITERALLY '00';
74 2 DECLARE N$CHNLS LITERALLY '5';

75 2 TRIGGER$MSG.LENGTH = SIZE(TRIGGER$MSG);
76 2 TRIGGER$MSG.TYPE = 65;
77 2 CALL RQSEND(.RET$TRIG,.TRIGGER$MSG);
78 2 CALL RQELVL(2);

```

```

79 2      DO FOREVER;
80 3      MSG = RQWAIT(.GO$PULSE,0);
81 3      CALL RQSEND(.RET$PULSE,MSG);
82 3      LOCK$MSG = RQWAIT(.T$PARAM$LOCK,0);
83 3      DO I = 0 TO N$CHNLS-1;
84 4      FIRST$CHANNEL$REGISTER = BOX$PARAMS(I).CHANNEL
      + ROL(GAIN,6);
85 4      COMMAND$REGISTER = GO$BIT
      OR EOC$INTERRUPT$ENABLE;
86 4      MSG = RQWAIT(.RQL2EX,0);
87 4      COMMAND$REGISTER = 0;
88 4      BOX$PARAMS(I).ACCUM = BOX$PARAMS(I).ACCUM
      + ADC$DATA$REGISTER;
89 4      END;
90 3      CALL RQSEND(.T$PARAM$LOCK,LOCK$MSG);
91 3      T$MSG = RQWAIT(.RET$TRIG,0);
92 3      CALL RQSEND(.TRIGGER,T$MSG);
93 3      END;
94 2      END ADC$TASK;
95 1      CONTROL$TASK:
96 2      PROCEDURE PUBLIC;
97 2      DECLARE (LOCK$MSG,T,MSG) ADDRESS;
98 2      DECLARE I BYTE;
99 2      DECLARE NCHNLS LITERALLY '5';
100 2      DECLARE TURN$LAMP$ON
      LITERALLY 'OUTPUT(0E7H)=SHL(I,1)';
101 2      DECLARE TURN$LAMP$OFF
      LITERALLY 'OUTPUT(0E7H)=SHL(I,1)+1';
102 2      DECLARE SETUP$8255 LITERALLY 'OUTPUT(0E7H)=80H;
      OUTPUT(0E6H)=OFFH';
103 2      SETUP$8255;
104 2      DO FOREVER;
105 3      MSG = RQWAIT(.TRIGGER,0);
106 3      CALL RQSEND(.RET$TRIG,MSG);
107 3      LOCK$MSG= RQWAIT(.T$PARAM$LOCK,0);
108 3      DO I = 0 TO NCHNLS-1;
109 4      BOX$PARAMS(I).COUNT = BOX$PARAMS(I).COUNT + 1;
110 4      IF BOX$PARAMS(I).COUNT
      = BOX$PARAMS(I).SAMPLES THEN
111 4      DO;
112 5      T,
      BOX$PARAMS(I).READING
      = (BOX$PARAMS(I).ACCUM
      /BOX$PARAMS(I).SAMPLES) / 38
      + BOX$PARAMS(I).OFFSET;
113 5      IF T <= BOX$PARAMS(I).SET$POINT
      - BOX$PARAMS(I).ERROR THEN
114 5      TURN$LAMP$ON;
      ELSE
115 5      IF T >= BOX$PARAMS(I).SET$POINT
      + BOX$PARAMS(I).ERROR THEN

```

```
116 5          TURN$LAMP$OFF;  
          BOX$PARAMS(I).ACCUM,  
          BOX$PARAMS(I).COUNT = 0;  
118 5          END;  
119 4          END;  
120 3          CALL RQSEND(.T$PARAM$LOCK,LOCK$MSG);  
121 3          END;  
122 2          END CONTROL$TASK;  
123 1          END ATOD;
```




APPLICATION NOTE

AP-47

I. INTRODUCTION	2-35
II. OVERVIEW	2-35
FORTRAN-80	2-35
Software Decisions	2-35
III. USING FORTRAN-80	2-36
I/O Capabilities	2-36
Math Capabilities	2-38
IV. APPLICATION EXAMPLE	2-39
An Automated Test Stand	2-39
V. USING THE ISBC RM	2-42
RMX\80™ Overview	2-43
The RMX\80™ Model	2-43
VI. APPLICATION EXAMPLE	2-44
A Sewage Treatment Plant Control System	2-44
VII. SUMMARY	2-50
APPENDIX A	2-51
APPENDIX B	2-53

November 1978

Using FORTRAN-80 for ISBC™ Applications

Steve Vertheye
OEM Microcomputer Systems Applications

Using FORTRAN-80 for iSBC™ Applications

Contents

I. INTRODUCTION	2-35
II. OVERVIEW	2-35
FORTRAN-80	2-35
Software Decisions	2-35
III. USING FORTRAN-80	2-36
I/O Capabilities	2-36
Math Capabilities	2-38
IV. APPLICATION EXAMPLE	2-39
An Automated Test Stand	2-39
V. USING THE iSBC 801	2-42
RMX/80™ Overview	2-43
The RMX/80™ Model	2-43
VI. APPLICATION EXAMPLE	2-44
A Sewage Treatment Plant Control System	2-44
VII. SUMMARY	2-50
APPENDIX A	2-51
APPENDIX B	2-63

I. INTRODUCTION

In March of 1978, Intel announced the availability of a resident FORTRAN compiler for the Intellec® Micro-computer Development System. In November of 1978, Intel announced the availability of a run-time package to support the execution of FORTRAN-80 compiled programs in the RMX/80™ environment. With this support package, user's of Intel's complete line of iSBC™ Single Board Computer products can benefit from the full set of I/O and math capabilities provided by the FORTRAN-80 language.

This application note is intended to familiarize the reader with the features, benefits and usage of the FORTRAN-80 package and RMX/80™ Executive. The reader who is unfamiliar with any of these topics is urged to refer to the related Intel publications listed in the front-piece.

Following the overview, two application examples will be studied. In the first example, FORTRAN code is used in a "stand-alone" environment; i.e., without operating system support. The second example is a multitasking system managed by the RMX/80 Executive which supports standard I/O interfaces to the RMX/80 Terminal Handler and Disk File System.

II. OVERVIEW

Intel's FORTRAN-80 compiler is an implementation of the standard FORTRAN known as ANS FORTRAN 77 approved by the American National Standards Institute (ANSI) in April, 1978. The implementation is of the FORTRAN 77 subset, plus most of the full I/O capability and Intel defined extensions. For a fuller description

of the implementation, consult the *FORTRAN-80 Programming Manual*.

FORTRAN-80 is a high-level applications programming language with flexible I/O handling and floating-point math instructions. With the FORTRAN-80 language, the programmer can easily implement sophisticated applications involving scientific calculations, process and instrument control, test and measurement, and a host of other applications requiring the power and flexibility the FORTRAN-80 language provides.

With the addition of the iSBC 801 FORTRAN-80 RUN-TIME PACKAGE for RMX/80 SYSTEMS, the user who wishes to implement his application using Intel's Single Board Computers and the RMX/80 Real-Time Multitasking Executive can take full advantage of the FORTRAN-80 I/O and math capabilities. The package allows the user to accelerate the run-time execution of FORTRAN-80 coded mathematical formulae through special interfaces to the optional iSBC 310™ High Speed Mathematics Unit. All disk and terminal I/O is interfaced directly to the RMX/80 Disk File System and either the full or the minimal Terminal Handler. The libraries that comprise the iSBC package are constructed in a modular fashion, allowing the user to configure systems with as much or as little of the support libraries as needed for a given application.

In order to effectively utilize the hardware and software products now available, it is important to design the application system from the top down. This implies that we need to think of an application in very general terms and then successively introduce more detail until we have program code as our final step. At each stage of the definition, we have to make decisions about the usage and configuration of various products.

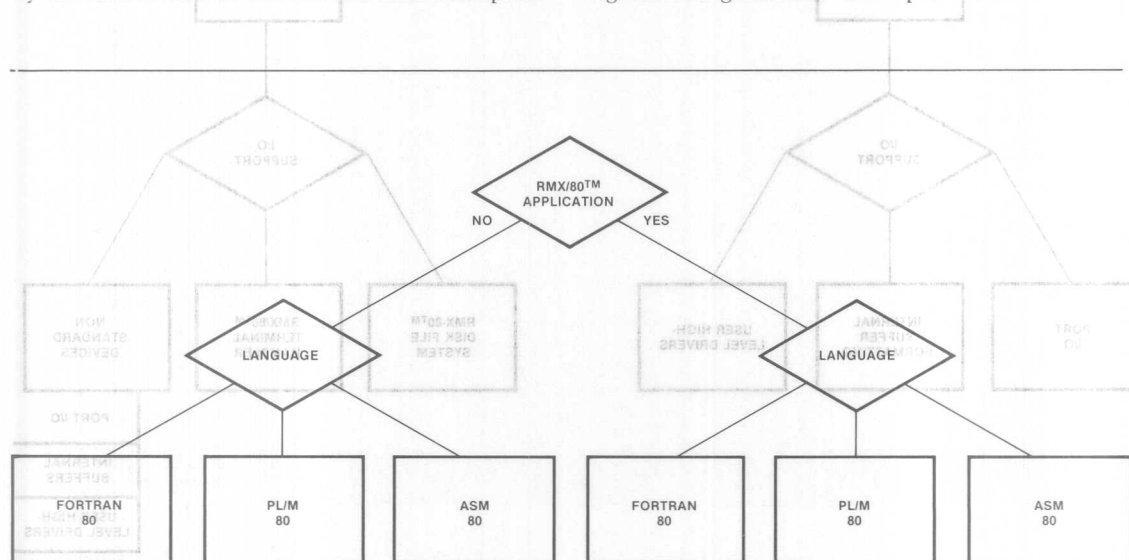


Figure 2. The I/O Support Decision

The decision-making process that concerns itself with software can be shown as a tree (Figure 1). The first decision that must be made is whether or not the RMX/80 Real-Time Multitasking Executive should be utilized. In general, this package will prove extremely useful if the application to be designed must respond to multiple asynchronous events, or contains multiple, semi-independent processes that could be executed in parallel, or has need of standard vendor supplied device drivers. If the application is very small and simple, handles few or no interrupts, has no need for parallel execution of multiple processes, and the designer is willing to supply his own I/O device drivers, the program may be able to execute without the support of an operating system.

Whether the RMX/80 package is used or not, the system designer must now choose in which language or languages the programs should be coded. Each of the three languages shown is optimized for different purposes. The PL/M-80 language is well suited for systems programming. The ASM-80 language is best suited for applications requiring direct control of the computer (e.g., the registers and memory). The FORTRAN-80 language is highly desirable for those applications requiring mathematical calculations and formatted

I/O. In many cases, the optimal solution will use a mix of two or even all three of these languages.

III. USING FORTRAN-80

I/O Capabilities

After the decision has been made to use the FORTRAN-80 language for an application, various types of I/O support are available to the user (see Figure 2). If the program code is to run without any support from an operating system, the user must supply drivers for any devices he wishes to include in his system.

When designing an RMX/80 system, the iSBC 801 package supplies the standard interface to the disk and terminal while the user may support additional devices in the same manner as the "stand alone" program would. The following sections expand on the topic of FORTRAN-80 I/O support.

Port I/O

The simplest and most direct method of performing I/O in the FORTRAN-80 language uses two pre-defined subroutines, INPUT and OUTPUT. The example below illustrates the use of these subroutines to input bytes from and output bytes to any of the 8080A/8085A I/O ports.

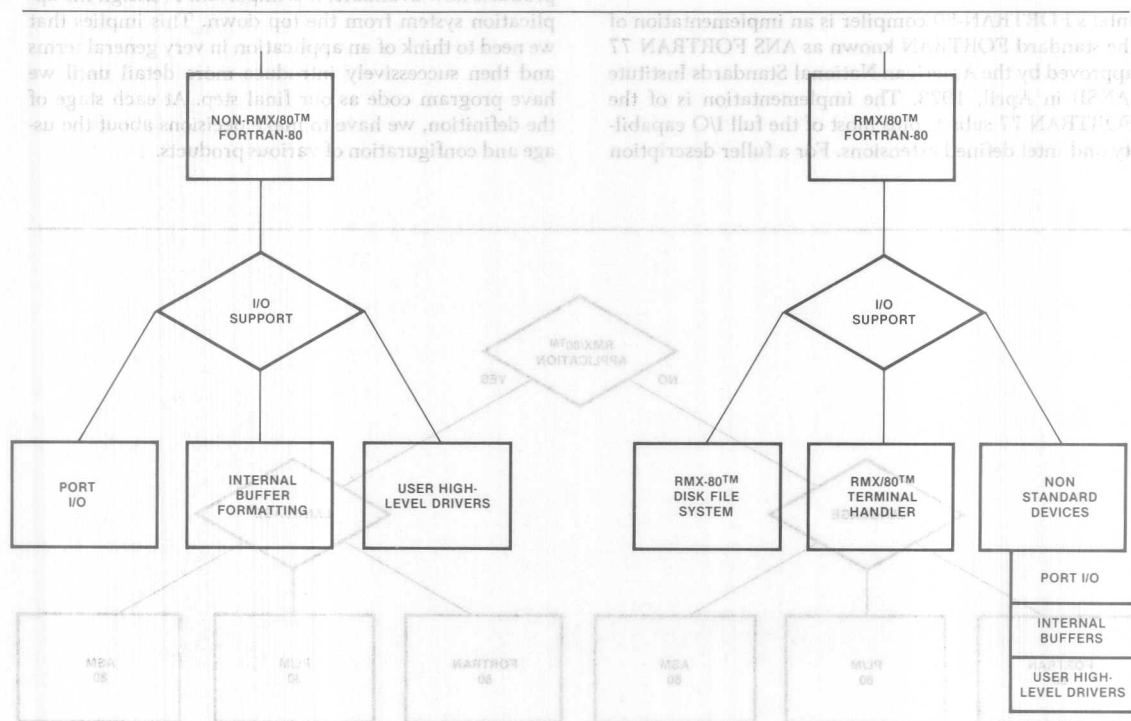


Figure 2. The I/O Support Decision

INTEGER*1 IVAL

C -- PROGRAM THE 8255 PARALLEL I/O CHIP

C -- PORT # = EB; VALUE = 94

CALL OUTPUT (#0EBH, # 94H)

C -- INPUT 8 BITS FROM PORT A INTO IVAL

C -- PORT # = E8; VALUE INPUT TO IVAL

CALL INPUT (#0E8H, IVAL)

Port I/O is extremely useful in many applications. However, this method requires the programmer to directly handle each and every byte. Also, it does not utilize the power of the FORTRAN-80 formatting routines.

Internal Buffer Formatting

One method of overcoming the shortcomings of Port I/O is to use character strings as "virtual devices." This is accomplished by specifying a character string as the unit number in a READ or WRITE statement. External routines can be used to fill the character strings with input for the READ statement and to output buffers that have been formatted by the WRITE statement. The example below shows the use of this method.

SUBROUTINE EXAMPL CHARACTER*80 BUFFER

C -- CALL DEVICE DRIVER TO GET BUFFER OF
C -- CHARACTERS

CALL BUFIN (BUFFER)

C -- NOW READ FROM BUFFER INTO VARIABLES
C -- UNDER FORMAT CONTROL

READ (BUFFER, 100) X, Y, Z
100 FORMAT (F10.3, F12.4, F13.5)

• PROCESS DATA STORED IN VARIABLES
• X, Y, Z

C -- WRITE RESULTS TO BUFFER

WRITE (BUFFER, 200) A, B, C, D
200 FORMAT (4F12.3)

C -- CALL DEVICE DRIVER TO OUTPUT BUFFER

CALL BUFOUT (BUFFER)

User Provided High-Level Drivers

If an application requires only simple input (READ) and output (WRITE) capabilities, the previous method would probably be sufficient. If, however, the device(s) in the system are more complex, it may be necessary to perform other I/O operations. One way of doing this would be to write subroutines for each operation. A much nicer solution is to use the FORTRAN-80 I/O instructions (OPEN, CLOSE, READ, WRITE, PRINT, BACKSPACE, REWIND, and ENDFILE) to interface to user-written routines which implement these instructions for the special device.

This is possible because, for each open file in the system, the FORTRAN-80 I/O system keeps a table connecting the unit number with the addresses of the routines that handle all operations on that unit. The I/O system allows the user to substitute his own device drivers into this table. To do this, the system designer codes a routine and labels it FQ0LVL. This routine is then made known to the I/O system (i.e., declared PUBLIC). Whenever a file is first accessed (i.e., OPENED), the I/O system calls FQ0LVL with a set of parameters, one of which is the file name referenced in the OPEN statement. The designer, in his code for FQ0LVL, scans the file name to decide if this is one of the files for which he wishes to supply drivers. If so, he passes back a table of the addresses of the routines that will take care of the eight primitive file I/O capabilities (refer to the example following this paragraph and to the *FORTRAN-80 Compiler Operators Manual*).

FQ0LVL: PROCEDURE(fileSptr, bufSptr) BYTE PUBLIC;

/* table of entry point addresses for driver routines */

DECLARE table (8) ADDRESS DATA

.openShdlr, /* address of OPEN routine */
.closeShdlr, /* address of CLOSE routine */
.readShdlr, /* address of READ routine */
.writeShdlr, /* address of WRITE routine */
.backShdlr, /* address of BACKSPACE routine */
.mv2recShdlr, /* address of MV2REC routine */
.rewindShdlr, /* address of REWIND routine */
.makeSeofShdlr /* address of END OF FILE routine */

);

DECLARE (returnedSstatus, index) BYTE;

DECLARE (fileSptr, bufSptr) ADDRESS;

DECLARE buf BASED bufSptr (1) BYTE;

DECLARE fileName BASED fileSptr (1) BYTE;

DECLARE analogSin (*) BYTE DATA('AI:');

/* set flag initially =FFH */

returnedSstatus=0FFH;

/* if any character of fileName does not compare set flag=0 */

DO index=0 TO 3;
IF fileName(index) <> analogSin(index) THEN
returnedSstatus=0;
END;

/* if flag=FFH pass back the addresses of the drivers */

IF returnedSstatus=0FFH THEN
CALL move(size(table), table, bufSptr);
RETURN returnedSstatus;
END; /* of FQ0LVL */

RMX/80™ Support

When using the RMX/80 Executive, the iSBC 801 FORTRAN-80 RUN-TIME PACKAGE¹ for RMX/80 SYSTEMS can be used to provide a direct interface to standard RMX/80 high level drivers, the Disk File System and the Terminal Handler. With the RMX/80 Executive, users can code multiple, concurrently executing programs that perform formatted I/O to disk files and the console, as shown in the following example:

```
C
C-- OPEN disk file
C
  OPEN (8,FILE = 'D0:TSTDAT.FIL',ACCESS =
    'SEQUENTIAL')
C
C-- perform tests
C
  .
  .
  .
C
C-- WRITE results to file for archival storage
C
  WRITE(8,100) (RESULT (I),I=1,10)
100 FORMAT(10F12.3)
C
C-- PRINT completion message on console
C
  PRINT 200
200 FORMAT ('TESTS COMPLETE')
  .
  .
  .
```

If it is necessary for a FORTRAN program in the RMX/80 system to perform I/O to a device not handled by one of the high level drivers, any of the methods previously described can be utilized to augment the I/O system.

FORTRAN-80 Math Capabilities

The FORTRAN-80 language supports four data types labelled INTEGER, REAL, LOGICAL, and CHARACTER. Also supported are various operators which can manipulate objects of various types. Both INTEGER (fixed point) and REAL (floating-point) objects can be manipulated by the add (+), subtract (-), multiply (*), divide(/), and exponentiation (**) operators. In addition, Integers can be operated on by the Boolean operators (e.g., .AND.,.OR.). In this case, the operations are performed bit-wise on the operands.

All floating-point arithmetic operations are performed with algorithms that adhere to the Intel Floating-Point Standard¹ which allows for seven decimal digits of precision. Whenever math operations are used, the user

can make the decision to use a software package to implement the floating point support or to accelerate the execution of these operations (by as much as a factor of five or six) by installing an iSBC 310 High-Speed Mathematics Unit and linking in special FORTRAN-310 drivers. In either case, due to the adherence to the standard, the results of all calculations will be identical. In addition, the libraries have been designed to allow the switch to be made from software routines to a faster hardware solution with *no* code changes.

Above and beyond the basic mathematical operators in FORTRAN-80, a large number of intrinsic functions are available. These functions provide services like type conversion, remaindering, and logarithmic and trigonometric calculation. Since the calculations involved in performing these high-level functions require the mathematical operators, they too can be accelerated by the inclusion of the iSBC 310 board and its associated drivers.

Error Handling

The math processing system also provides flexible error handling. The user can choose to use either an Intel-supplied error handler or one of his own design. The capability also exists to change the active error handler dynamically in cases where different routines require different handlers. The default error handlers are named FQFERH. One exists in each of the arithmetic libraries (Figure 3). This error handler will attempt to recover from an error by taking the most reasonable action (e.g., underflow error returns result=0). If code is being run "stand-alone" or under the RMX/80 executive the handlers in the math libraries should be used or the user should supply his own. Appendix B of the *ISIS-II FORTRAN-80 Compiler Operator's Manual* contains all of the information necessary to implement a custom error handler or to use the default routines.

FPF0FT.LIB	- Software package for "stand-alone" and ISIS-II systems
FPFARD.LIB	- iSBC 310 drivers for same
*FPF0TX.LIB	- Software package for RMX/80 systems
*FPF0RDX.LIB	- iSBC 310 drivers for iSBC 80/20, 80/20-4 and 80/30 boards
*FPF0X10.LIB	- iSBC 310 drivers for iSBC 80/10 and 80/10A boards
FPF0F.LIB	- Library of routines implementing intrinsic functions

*Available in iSBC 801 FORTRAN-80 RUN-TIME PACKAGE for RMX/80 Systems.

Figure 3. Available Math Libraries

¹ Palmer, John F., "The Intel Standard for Floating-Point Arithmetic," *Proceedings of the First International Computer Software and Applications Conference* (Chicago: IEEE Computer Society), November, 1977, pp 107-112.

IV. APPLICATION EXAMPLE

An Automated Test Stand

This example shows the steps taken to design and implement an automated test stand. The hardware system must interface to a test fixture upon which test items can be mounted. Operator inputs and test outputs involve a 300-baud hard copy terminal. The software to be developed must allow an operator to invoke a variety of tests from the console and to receive some printed performance record for the object under test. In addition, the software must allow for tests to be added and deleted often, and each test must be allowed to obtain any number of parameters from the command line tail.

After examining the problem definition and the decision making diagram presented earlier, it was decided that this application could be implemented with a simple sequential program.

Since formatted I/O and mathematical calculations are involved, the FORTRAN-80 language is well suited to be the main programming language. Also, some ASM-80 routines will come in handy for communicating with the console.

An analysis of the I/O to be performed breaks down into two distinct types. Various inputs to and outputs from the test fixture will be 8-bit parallel transfers. These will likely go through the 8255A ports on the Single Board Computer. Port I/O will be used to handle this function. Interface with the operator requires READ'S AND WRITE's to the console device. The simplest way of performing this function is to use character strings as the target of READ and WRITE operations and coding small ASM-80 routines to transfer these buffers from/to the console.

A diagram of the test stand is shown in Figure 4. The computer hardware necessary to solve this application includes a Single Board Computer (the iSBC 80/20 board), a PROM memory module and an analog I/O board. Digital I/O with the test fixture is handled by the 8255A ports on the Single Board Computer. The analog inputs on the test fixture come from the two D/A converter channels on the iSBC 732 board.

The software solution utilizes a very rudimentary command line interpreter. The mainline routine gets a line of input and finds the first non-blank character. If this character is an alphabetic character, it is used in a computed GOTO statement to transfer control to one of a possible 26 entry points. Tests may be added by choosing a keyletter and inserting a label in the GOTO statement to transfer control to the new test routine. The command input line and the index in the line are stored in a common block so that any test routine can continue scanning the line for parameters or can reset the index and find out what keyletter caused its invocation. The flow of the software is illustrated in Figure 5.

For the purpose of explanation, routines are shown to implement a "calculator mode" which allows the opera-

tor to perform arithmetic from the console, and a logic transition tester which determines whether the object on the test fixture changes state at the proper voltages.

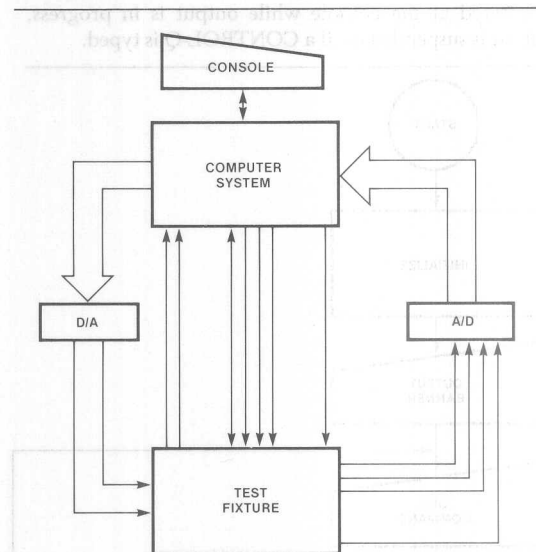


Figure 4. Test Stand Diagram

Code Description

The following sections describe the program code for this application example. Fold-out code listings are contained in Appendix A. The circled reference letters in the text refer to the corresponding letters in the listings.

The DRIVRS Module

The module DRIVRS contains three primary routines. START (A) is located at 0 so that it is executed upon power up. This routine is responsible for programming the on-board hardware (8255A, 8251, 8253), setting up the system stack, and calling the FORTRAN routine labeled MAINLN.

The input routine BUFIN (B) is called from FORTRAN routines with a character string as an argument. Note that passing a string argument from FORTRAN results in the address and length of the string being sent as parameters. The string is filled with characters input from the console until a carriage return is encountered. A simple line-editing scheme is implemented allowing character deletion (RUBOUT), line deletion (CONTROL-X), and echoing of the current buffer contents (CONTROL-R). Attempted entry of characters beyond the end of the string and RUBOUTS past the beginning cause the audible bell to sound.

The output routine, BUFOUT ^(C), also takes a character string as an argument. The entire contents of the string are sent to the console unless a carriage return is encountered in the string. If a carriage return is the terminator, a line feed is output as well. If a CONTROL-S is entered at the console while output is in progress, output is suspended until a CONTROL-Q is typed.

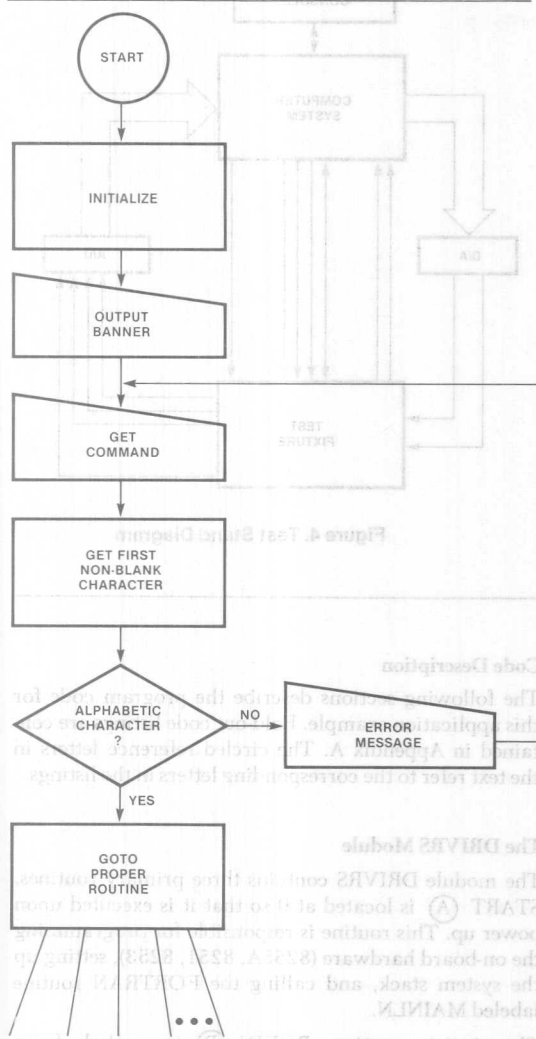


Figure 5. Flow Diagram

The MAINLN Module

The module MAINLN ^(D) contains the mainline routine that implements the command line interpreter. The statement IMPLICIT LOGICAL A-Z will cause most usages of undeclared variables to be reported as illegal mixed mode; the intent in writing these programs was to declare all variables, which is generally considered

good programming practice, even though Fortran makes default assumptions about undeclared variables.

The default handler is to be used for any errors that may occur while performing mathematical calculations. Also, the routines that perform the calculations must be initialized. Both of these operations are performed by the call to FQFSET ^(E). The call takes two arguments. The first argument is a two byte field specifying which error handler is to be used. If the low order bit of the high order byte is a one (e.g., 100 hexadecimal), the math routines will call a user error handler whose address is given as the second parameter. If the low order bit is zero (as is the case in this example), the routines will use the default handler and ignore the second argument.

A banner is output to the console by the sequence at ^(F) where a formatted WRITE is performed on an internal buffer (IMAGE) and then the external driver BUFOUT is called to output the buffer to the console. The variable CARRET is used to insert a carriage return into the string to be output. In order to allow individual characters in the character string to be accessed, the EQUIVALENCE statement is used to cause LINBUF and IMAGE to occupy the same memory space. The variable INDEX ^(G) is used to scan through the input buffer.

A call to BUFIN ^(H) fetches the command line from the console. DBLANK is called ^(I) to position INDEX to the first non-blank character. This character is converted to its integer representation, normalized to 1 and checked to see if it is a valid alphabetic character ^(J). If the keyletter is valid, the computed GOTO ^(K) causes execution to branch to the correct point in a jump table ^(L). Note that A (add.), S (subtract), M (multiply), and D (divide) all branch to a single routine MATH, T (transition test) branches to a routine called TRANST and all other keyletters are trapped into line 100. Any and all I/O errors cause the ERROR routine to be called.

The DBLANK Module

The DBLANK routine ^(M) de-blanks the input line. If a carriage return is encountered, the operator is prompted for more input.

The ERROR Module

The ERROR routine ^(N) prints out an error message, with the error number, to the console.

The MATH Module

In many of the tests, the human operator must supply numeric parameters. A calculator mode is supplied for the simple calculations that might be needed here. This mode is implemented through the MATH routine ^(O). Since any one of four keyletters could have caused this routine to be invoked, MATH rescans the command line to obtain the keyletter ^(P). Following this, two operands are read in by calls to CONVRT ^(Q) and the operation requested is performed on them.

The CONVRT Module

Subroutine CONVRT [®] is called from other routines to extract floating-point operands from the input line buffer. Characters are transferred into a temporary buffer [®] until either a carriage return or a comma is encountered. The temporary buffer is then read under format control to obtain the returned value [®].

The TRANST Module

The item to be tested is composed of combinatorial logic as shown in Figure 6. The transition test sets all inputs except one to a constant value. By varying the voltage at the remaining input, the transitions at the output can be checked. This test must be run while the +5V power to the fixture is varied through a range of values. This testing is performed by the TRANST routine.

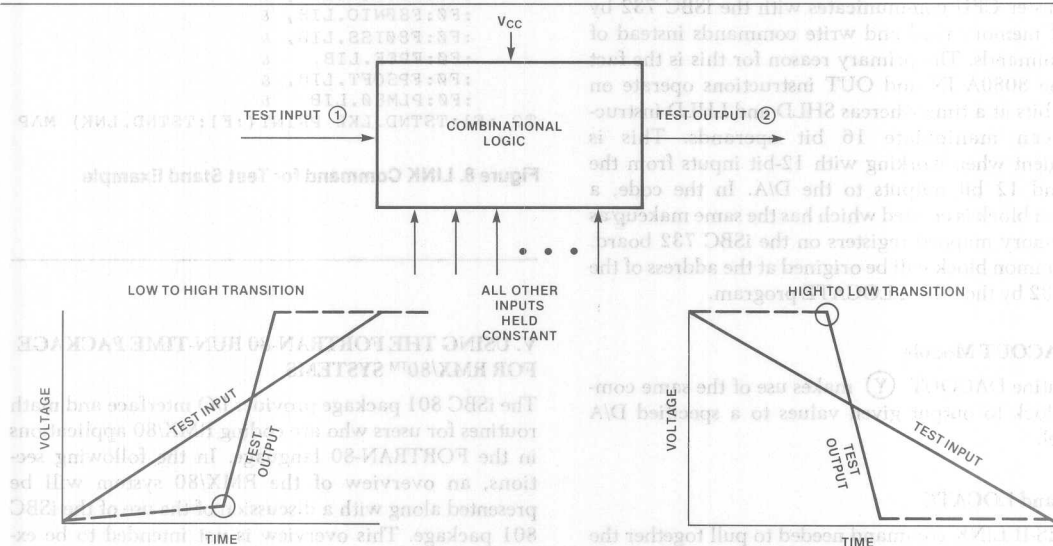


Figure 6. Transition Test

```
TEST STAND V0.0
COMMAND?
M 34.678,345.43
      34.67800 *      345.43000 =      11978.82160
COMMAND?
T 4.5,5.5,.2,5.
TRANSITION TEST TOLERANCE= 5.0% AMBIENT TEMPERATURE = 25.30 DEGREES C
VCC  HIGH TRANS  LOW TRANS  HIGH  LOW  TEST
4.5  00.81  3.42  4.43  0.12  PASSED
4.7  00.80  3.44  4.67  0.08  PASSED
4.9  00.80  3.67  4.88  0.02  PASSED
5.1  00.80  3.71  4.93  0.02  PASSED
5.3  00.80  3.73  4.98  0.01  PASSED
5.5  00.80  3.74  4.99  0.01  PASSED
COMMAND?
```

Figure 7. Sample Output

Power is supplied to the test fixture through one of the two D/A channels on the iSBC[™] 732. Three of the input parameters specify the starting and stopping voltage values for Vcc and the increment to be added each step. The fourth parameter is the tolerance to be used to decide if the test passes or fails at each step. Once the test is running, the output voltage at [®] is measured for inputs at [®] of 0 and 5V. The voltage input is then incremented from 0V (using D/A channel 1) until a transition is sensed in the output voltage at [®]. At this point, the input voltage at [®] is checked to see if it is within tolerance. The same process is then repeated with the voltage at [®] going from +5V downward. After the test is complete, a formatted report is generated containing the ambient temperature (measured through a temperature sensor) and the performance record for the item under test.

An external routine (U), ADCIN, is called to input samples into the variable given as the first parameter from the channel given as the second parameter. The counts from the temperature sensor exhibit a logarithmic curve, so the input is linearized using the equation shown. The routine DACOUT (V) takes the first parameter and outputs it to the channel specified by the second parameter. If no transition occurs when the test input is run through its entire range, the item is assumed non-functional, a message is output, and control is returned to the console (W).

The ADCIN Module

Subroutine ADCIN (X) fetches samples from the A/D converter on the iSBC 732 board. The channel number is an input parameter and the data is the returned value. Of special note in this routine is the use of the FORTRAN common block to control a memory-mapped device. The master CPU communicates with the iSBC 732 by way of memory read and write commands instead of I/O commands. The primary reason for this is the fact that the 8080A IN and OUT instructions operate on only 8 bits at a time whereas SHLD and LHLD instructions can manipulate 16 bit operands. This is convenient when working with 12-bit inputs from the A/D and 12 bit outputs to the D/A. In the code, a common block is created which has the same makeup as the memory mapped registers on the iSBC 732 board. The common block will be originated at the address of the iSBC 732 by the ISIS-II LOCATE program.

The DACOUT Module

Subroutine DACOUT (Y) makes use of the same common block to output given values to a specified D/A channel.

LINK and LOCATE

The ISIS-II LINK command needed to pull together the individual pieces of this example is shown in Figure 8. After compilation, the object modules of all of the previously described routines are placed in the library FRTMOD.LIB by the ISIS-II Library Manager™. The LINK statement starts with the module DRIVRS.OBJ, which has one EXTERNAL reference, MAINLN. To

satisfy this reference, MAINLN.OBJ is linked in from FRTMOD.LIB and its EXTERNAL references cause the inclusion of other modules.

The LOCATE command shown in Figure 9 is used to assign absolute memory locations to the code in the LINKED modules. The common block labelled ADC is explicitly assigned to FFF0H so that it will correctly overlay the memory-mapped space of the iSBC 732 board. The ORDER statement is used to tell the locator in what order the various segments should be placed in memory.

```
LINK :F1:DRIVRS.OBJ, &
      :F1:FRTMOD.LIB, &
      :F0:F80RUN.LIB, &
      :F0:F80NIO.LIB, &
      :F0:F80ISS.LIB, &
      :F0:FPEF.LIB, &
      :F0:FPSOFT.LIB, &
      :F0:PLM80.LIB &
TO :F1:TSTND.LK0 PRINT(:F1:TSTND.LNK) MAP
```

Figure 8. LINK Command for Test Stand Example

V. USING THE FORTRAN-80 RUN-TIME PACKAGE FOR RMX/80™ SYSTEMS

The iSBC 801 package provides I/O interface and math routines for users who are coding RMX/80 applications in the FORTRAN-80 language. In the following sections, an overview of the RMX/80 system will be presented along with a discussion of the use of the iSBC 801 package. This overview is not intended to be exhaustive. If the reader is unfamiliar with the RMX/80 package, he should gain from this section enough understanding to comprehend the concepts in the example presented. If the reader is planning on implementing an RMX/80 system, the RMX/80 references in the front-piece should be studied carefully.

```
LOCATE :F1:TSTND.LK0 PRINT(:F1:TSTND.LOC) MAP LINES SYMBOLS PUBLICS &
ORDER(CODE DATA /LINE/ /ADC/) /ADC/((0FFF0H) STACKSIZE(0) CODE(0)
```

Figure 9. Locate Command for Test Stand Example

Overview of the RMX/80™ Executive

A large number of microcomputer applications require the ability to respond to events in real-time. The RMX/80 Executive provides the system software around which you can build a real-time multitasking application using Intel iSBC 80™ Single Board Computers. In addition, the RMX/80 package provides the application designer with various high-level drivers (such as a terminal handler and a disk file system) which make it easier to develop sophisticated applications software.

The RMX/80™ Model

At this time, it is appropriate to discuss the RMX/80 model, or in other words, the general concepts upon which the RMX/80 Executive is built. Real-time systems, such as the RMX/80 system, provide the capability to control and respond to events occurring asynchronously in the physical world. To handle these events, the application is broken up into smaller semi-independent pieces, and each of these pieces is brought into action to handle the event for which it is intended. Each of these independent program units is a *task*. The RMX/80 Executive manages the execution of these tasks in accordance with a user-designated priority scheme to insure that the highest-priority task in the system has control of the CPU. It is also necessary, in a system such as this, for these semi-independent program units (tasks) to communicate with each other. This communication may be for the purpose of synchronization, data passing, mutual exclusion or any other use that may arise. To facilitate inter-task communication, the RMX/80 model incorporates the notion of messages and exchanges. A *message* is a data structure that can contain an arbitrary amount of information to be communicated from one task to another. An *exchange* is a "mail box" where tasks may send messages to be picked up by other tasks. The primary operations (primitives) that accomplish message transfers in the RMX/80 system are RQSEND* and RQWAIT*. Figure 10 diagrammatically shows the interaction of tasks, messages, and exchanges and introduces the symbolism used to represent these RMX/80 concepts in the system design.

Tasks

Typically, a task will execute a section of code that performs some initialization and then enters an infinite loop performing some processing over and over again as shown in Figure 11.

Each task in the system has a priority associated with it. The RMX/80 Executive uses this priority scheme to determine which ready task to run. The assignment of priorities to individual tasks is up to the system designer, giving him the capability to tune his system by assuring timely execution of important functions.

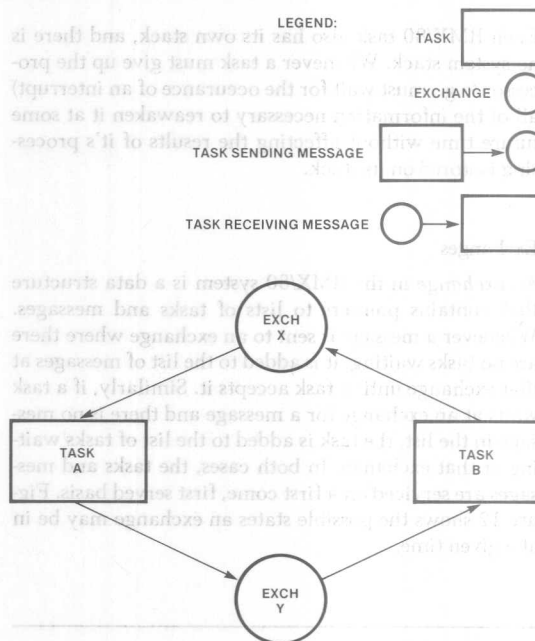


Figure 10. Task, Message, Exchange Interaction

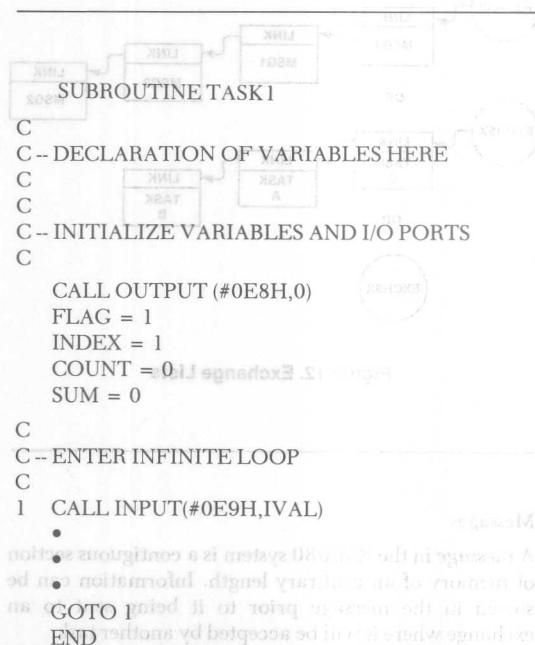


Figure 11. Task Loop

*In order to differentiate RMX/80 procedures and data structures from the user's, the names of system objects are always preceded by RQ.

Each RMX/80 task also has its own stack, and there is no system stack. Whenever a task must give up the processor (e.g., must wait for the occurrence of an interrupt) all of the information necessary to reawaken it at some future time without affecting the results of its processing is stored on its stack.

Exchanges

An *exchange* in the RMX/80 system is a data structure that contains pointers to lists of tasks and messages. Whenever a message is sent to an exchange where there are no tasks waiting, it is added to the list of messages at that exchange until a task accepts it. Similarly, if a task waits at an exchange for a message and there is no message in the list, the task is added to the list of tasks waiting at that exchange. In both cases, the tasks and messages are serviced on a first come, first served basis. Figure 12 shows the possible states an exchange may be in at a given time.

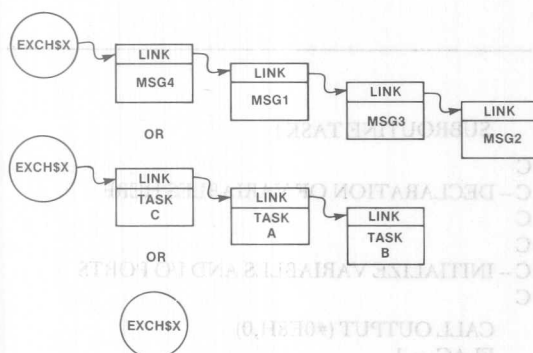


Figure 12. Exchange Lists

Messages

A *message* in the RMX/80 system is a contiguous section of memory of an arbitrary length. Information can be stored in the message prior to it being sent to an exchange where it will be accepted by another task.

Configuration

The configuration module contains various tables and PUBLIC variables that are accessed by the system at start-up time. All of the necessary information on the tasks and exchanges to be created and the disk file

system to be utilized are contained in this section of code. Configuration modules can be coded in either PL/M or assembly language (for which there are macros included with the RMX/80 product.)

Memory Usage

In systems using disk, it is necessary to ensure that certain buffers used by the disk controller for Direct Memory Access (DMA) are located in memory that is accessible to the disk controller. The buffers needed are allocated in a separate module called the *controller addressable memory* module. In the case of the iSBC 80/10, 80/10A, 80/20, and 80/20-4 boards, this module should be LOCATED before being included in the LINK statement to make sure that it does not contain any RAM on the CPU board itself (and, therefore, not controller-addressable). This restriction does not apply to iSBC 80/30 systems, since the iSBC 80/30 board has a dual port bus allowing system access to on-board RAM.

VI. APPLICATION EXAMPLE

A Sewage Treatment Plant Control System

In the early 1900's, the most popular type of sewage treatment system was known as a Sequencing Batch Reactor. It provided excellent effluent quality, but as populations grew, the amount of control necessary to operate the plant became too great for human operators, and a new type of treatment system came into use. This new system did not require such accurate control, but it also did not perform as well. With the passage of stricter and stricter water quality laws, and with the advent of low-cost, high powered microcomputer control systems, a serious look is being taken once again at Sequencing Batch Reactors.

A diagram of the treatment system and its sensors and actuators is shown in Figure 13. The system usually consists of three tanks, with each tank having individual influent and effluent valves, mixers and aeration equipment.

At any given time, all influent is being routed to one tank. When this tank is filled, the influent is routed to one of the other two. The full tank is agitated and aerated until the bacteria in the tank digest the sewage to within given limits. At this time, the mixer and aerator are turned off and the contents settle. After a time, the supernatant fluid is drawn off leaving the layer of concentrated bacteria to digest the next batch.

The computer control system necessary for controlling these reactors is shown in Figure 14. The system is responsible for monitoring the various sensors and contact closures, maintaining archives of system status, logging reports upon command, activating operator alarms, and performing on-line control of the batch cycle.

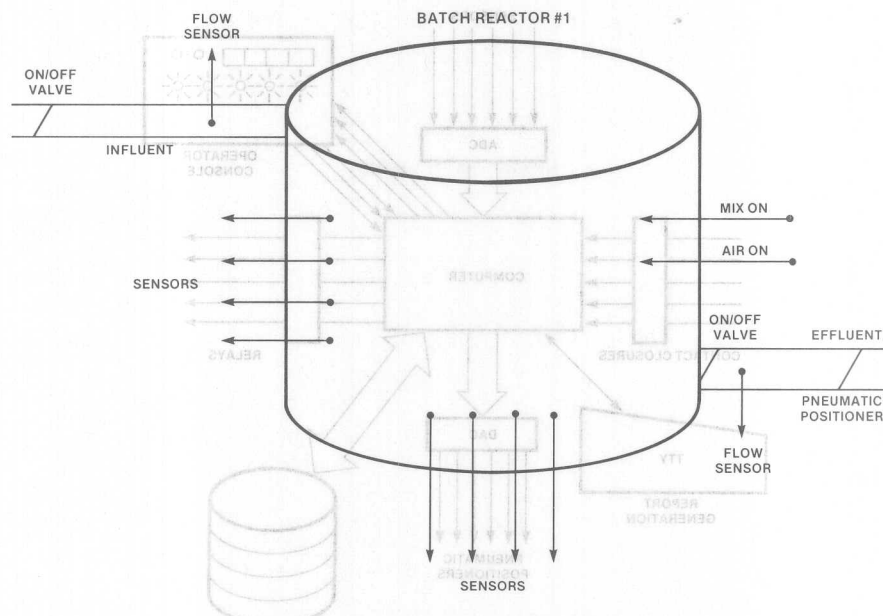


Figure 13. Sewage Treatment System and Sensors

Software

An analysis of the functions that need to be performed by the software for this control system leads to a decision to use the RMX/80 Executive. Timely response to multiple asynchronous events is the main thrust of this application. A breakdown of the individual functions in the system would be:

- **data collection** — gathers inputs from the sensors and contact closures and stores them where other routines can access them. Also, converts data from analog counts to engineering units.
- **on-line control** — based on current sensor inputs determines whether aeration, agitation, discharge and fill should be on or off.
- **alarm scanning** — compares current status values with setpoints and sets operator alarms if conditions are out of tolerance when effluent is on.
- **data logging** — once every five minutes logs current system status into a disk file record.
- **real-time clock** — maintains day, month, year, and time of day.
- **operator console handler** — monitors operator console to detect operator commands for time and set-point changes, report generation, alarm clearing, etc.
- **report generation** — upon operator command, formats either the file corresponding to yesterday's operation or today's operation to the current moment.

Each of these functions must be studied independently

before the decision on which language to use for each is made. The functions concerned with data collection, on-line control, and alarm scanning will be concerned with mathematical calculations. The functions concerned with data logging and report generation will have need of formatted disk and console I/O. These routines will thus be coded in the FORTRAN-80 language.

As was mentioned earlier, the PL/M-80 language is a systems programming language. This means that it is optimized to deal with the concepts embodied in a high-level system such as the RMX/80 system. The program code that implements the real-time clock and operator console handler will be written in the PL/M-80 language. In addition, various PL/M-80 support routines will be written to be called on by one or more of the FORTRAN-80 routines. The purpose of these routines will be explained as they come up in the code descriptions following.

Hardware

The hardware used to implement this control system must perform the following functions:

- inputting analog samples from the various sensors
- outputting analog values to the pneumatic positioners
- inputting digital values from the contact closures and operator console
- outputting digital values to the operator console and alarm panel
- storing and retrieving data from diskette files

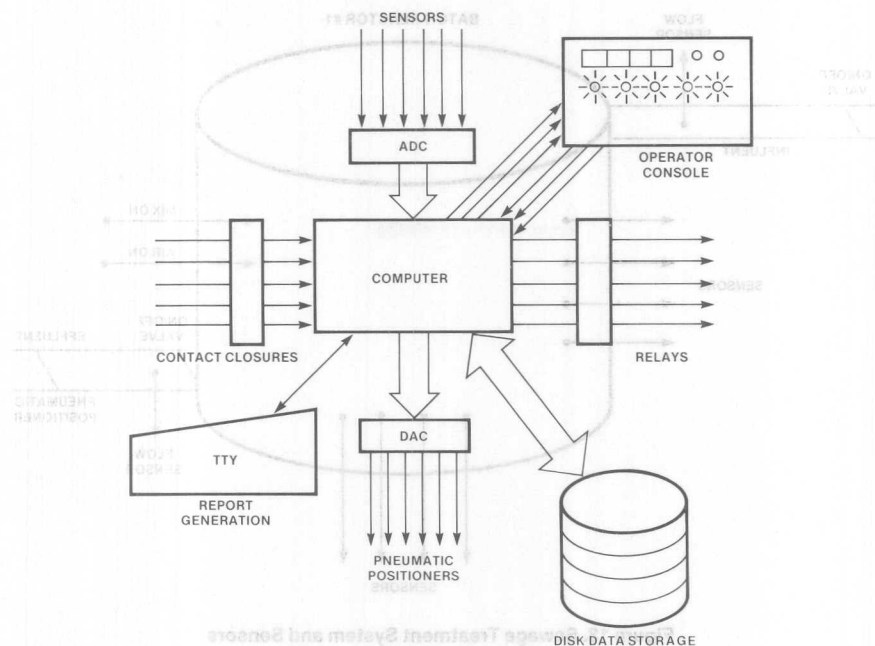


Figure 14. Computer Control System

The hardware configuration chosen includes an iSBC 80/30 Single Board Computer, an iSBC 732 Combination Analog Input/Output Board, a combination of PROM and RAM memory modules, and an iSBC 201 Diskette Controller.

There are various types of I/O devices in this system and each will require different FORTRAN-80 I/O support. The terminal and disk devices are supported through the iSBC 801 run-time package and the RMX/80 high level drivers. Communications with the A/D and D/A converters is accomplished using internal buffer formatting in conjunction with the RMX/80 Analog Handlers. Finally, port I/O is used for the digital inputs and outputs.

The next step in the design is to assign the system software functions to individual tasks in a manner that will allow for their parallel execution. The following tasks will be created to handle this application:

- STSINP — status input and unit conversion
- CNTRLO — on-line control
- SCAN and TIMER5 — alarm scanning and data logging
- TIMER and TIMUPD — real-time clock
- CONSOL — operator console handler
- REPORT — report generation

Figure 15 shows the interaction of these tasks in the RMX/80 system.

System Considerations

At this point, let us consider some of the mechanisms this system will require to synchronize and co-ordinate the tasks we have created. Status and setpoint information will be stored in FORTRAN common blocks. This will allow the STSINP, CNTRLO, SCAN, CONSOL, and TIMUPD tasks access to the STATUS information, and the CNTRLO, SCAN, and CONSOL tasks access to the SETPNT information. Once per five minutes, SCAN will be notified through a flag byte (MIN5UP) that he is to write the current system status to the file TODAYS.RPT. Upon command from the operator, REPORT will need to read these files to generate reports.

Since the RMX/80 system is designed to handle asynchronous events, it is quite possible for any of the tasks to be pre-empted at any point in their execution (e.g., an interrupt occurs or a higher priority task becomes ready to run). Thus, the SCAN task may be in the process of reading the last byte of a four-byte REAL integer when STSINP pre-empts the SCAN task and writes new information into the STATUS common block, thus invalidating the current SCAN operation. In another instance, REPORT may be in the process of fetching a disk record when SCAN attempts to write to the file. For these reasons, and more, it is necessary to implement some sort of synchronization mechanism in this system. We will insure that at most, one task has access to the common blocks and disk records by using a technique called *mutual exclusion*. In the RMX/80 system, this is accom-

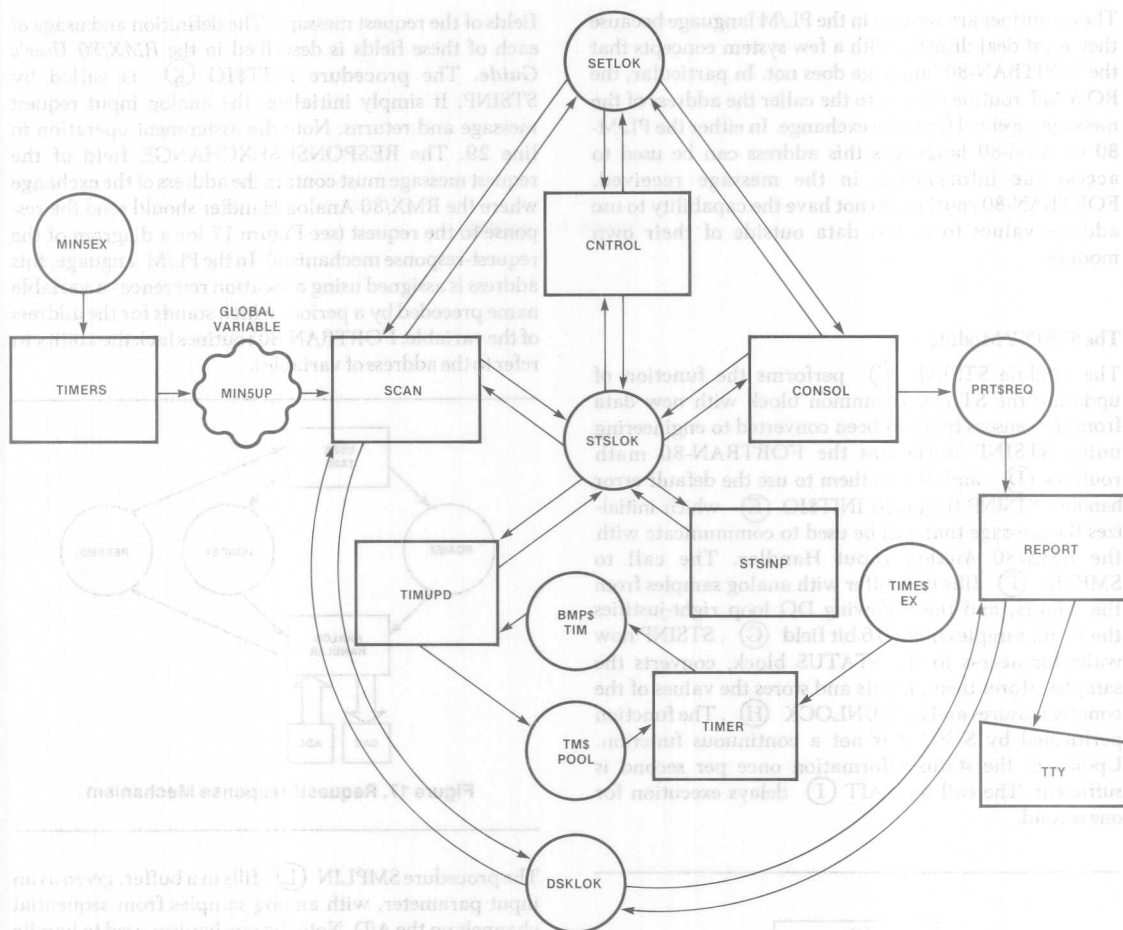


Figure 15. System Design Diagram

published by creating an exchange for each shared resource and initially sending one message to it. Any task requiring access to the resource first waits at the associated exchange for the key message. If a message is at the exchange, the task obtains the message and continues running until finished and then sends the message back. If another task waits at the exchange while the first is processing, it will stop execution until the first task finishes and returns the message.

Code Descriptions

What follows is a description of the code used to implement most of the tasks discussed. Appendix B contains fold-out code listings with circled reference letters. In the description, sections of the code will be called out using

circled letters that correspond to symbols in the appendix.

The Semmod Module

Two PL/M routines called LOCK and UNLOCK perform the mutual exclusion operation discussed earlier. There are three exchanges used for the purpose of exclusion: STSLOK, SETLOK, and DSKLOK. They govern access to the STATUS common block, the SETPNT common block and the disk file respectively. The LOCK procedure (A) takes one parameter, a number representing one of the three exchanges, and performs a wait at the appropriate exchange. Note the use of based variables to access the parameter. This is necessary since FORTRAN passes parameters by reference (address) rather than by value. The UNLOCK procedure (B) takes the same parameter and sends the single key (message) back to the appropriate exchange.

These routines are written in the PL/M language because they must deal directly with a few system concepts that the FORTRAN-80 language does not. In particular, the RQWAIT routine returns to the caller the address of the message received from the exchange. In either the PL/M-80 or ASM-80 languages this address can be used to access the information in the message received. FORTRAN-80 routines do not have the capability to use address values to access data outside of their own module.

The STSINP Module

The module STSINP ^(C) performs the function of updating the STATUS common block with new data from the sensors that has been converted to engineering units. STSINP initializes the FORTRAN-80 math routines ^(D) and directs them to use the default error handler. STSINP then calls INIT\$IO ^(E) which initializes the message that will be used to communicate with the RMX/80 Analog Input Handler. The call to SMPLIN ^(F) fills the buffer with analog samples from the sensors, and the following DO loop right-justifies the 12-bit samples in the 16 bit field ^(G). STSINP now waits for access to the STATUS block, converts the samples, stores them, inputs and stores the values of the contact closures and calls UNLOCK ^(H). The function performed by STSINP is not a continuous function. Update of the status information once per second is sufficient. The call to WAIT ^(I) delays execution for one second.

LINK	
LENGTH	
TYPE	
HOME EXCHANGE	
RESPONSE EXCHANGE	
STATUS	
BASE REGISTER POINTER	
ARRAY1 POINTER	
ARRAY2 POINTER	
COUNT	

Figure 16. Request Message for Sequential Channel Input with Single Gain

The ANALOG\$IO\$MOD Module

In the module labelled ANALOG\$IO\$MOD, the declaration of READ\$MSG ^(J) uses the predefined LITERAL called AI\$MSG. This LITERAL is one of many in the RMX/80 package that can be used to attach meaningful symbolic names to PL/M data structures. In this instance, AI\$MSG defines the fields of a standard analog input request message. Figure 16 is a diagram of the individual

fields of the request message. The definition and usage of each of these fields is described in the *RMX/80 User's Guide*. The procedure INIT\$IO ^(K) is called by STSINP. It simply initializes the analog input request message and returns. Note the assignment operation in line 29. The RESPONSE\$EXCHANGE field of the request message must contain the address of the exchange where the RMX/80 Analog Handler should send the response to the request (see Figure 17 for a diagram of the request-response mechanism). In the PL/M language, this address is assigned using a location reference - a variable name preceded by a period, which stands for the address of the variable. FORTRAN-80 routines lack the ability to refer to the address of variables.

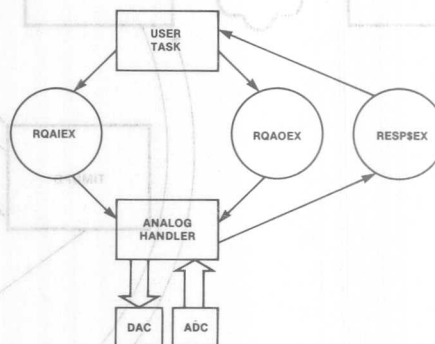


Figure 17. Request/Response Mechanism

The procedure SMPLIN ^(L) fills in a buffer, given as an input parameter, with analog samples from sequential channels on the A/D. Note the mechanism used to handle the passing of a FORTRAN string as a parameter. For every string in the parameter list, FORTRAN passes the starting address of the string followed by its length in bytes.

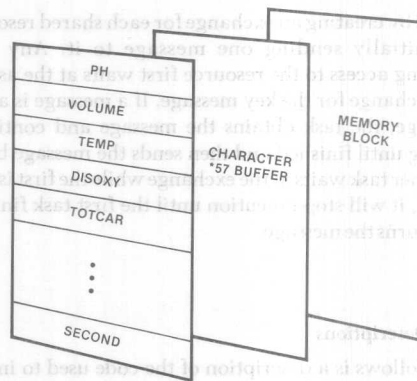


Figure 18. Use of EQUIVALENCE Statement

The SCAN Module

The SCAN task is responsible for operator alarms and data logging. The EQUIVALENCE statements (M) cause the STATUS common block to be overlaid by a character string, as illustrated in Figure 18. This allows for a compact file on disk of numerical data which can be broken out later for report generation.

After initialization, SCAN waits for access to both the STATUS and SETPNT common blocks. Operator alarms need to be set only if a parameter is out of specification and the effluent pump is on (N). After performing the scan, the variable MIN5UP is checked (C) to see if a report should be logged. If so, SCAN gains access to the disk file and writes a single record (P). All locks are now released, a one-second delay is counted out, and SCAN repeats the whole process. Normally, any errors that occur during the execution of I/O statements in the run-time package cause a message to be output on the console and the offending task to be suspended. Since this action is often undesirable, it is wise to handle one's errors programatically (Q).

The MIN\$5\$MOD Module

The module MIN\$5\$MOD contains two procedures. Both routines make use of the timed wait facility in the RMX/80 system. Any time a task calls RQWAIT to wait for a message at an exchange, an optional time limit (in 50 msec. intervals) can be specified. This is useful if the designer does not wish the task to be hung up forever if a message is never sent to that exchange. This mechanism can also be used to implement a timed wait if an exchange is specified to which no one will ever send a message. WAIT (R) delays execution of the calling task for one second. TIMER5 (S) waits for five minutes and then sets the variable MIN5UP to signal SCAN to log a disk record of current status.

The REPORT Module

The system console contains two buttons, one each for requests for printouts of today's and yesterday's status reports. Whenever one of these two buttons is pushed, the CONSOL task sends a message to the PRTREQ exchange with the TYPE field indicating which file to print. REPORT accepts these messages, checks the TYPE field (T), and calls the FORTRAN subroutine PRINT with the appropriate filename as a parameter. It then returns the request message to its sender via the response exchange field and waits for another request. Figure 19 is an example of the report generated by this system.

The PRINT Module

The PRINT subroutine will read in the compressed records written by SCAN and use the same set of EQUIVALENCE statements to break out the numerical data so

that it can be formatted for printout. If the type field (U) indicates that today's file is being accessed, PRINT obtains the key to the DSKLOK exchange since SCAN may disturb output operations if it attempts to log a new record. If yesterday's file is being accessed, the lock is not necessary, since no other task will be accessing this file. Once the lock is obtained, a record is read, the digital value of the contact closure status is converted to a more readable form (V) (ON or OFF), and the status line is formatted and printed. Since the SCAN task has an important function (operator alarms), we do not wish to hold it up for long if it happens to want to log a new status record. For this reason, PRINT relinquishes access to the file after every tenth record to allow SCAN to log its record and continue on. The rest of the code (W) checks for end of file indications and returns when printing is finished.

The INITMOD Module

Last in order (but first in execution) is the INIT procedure. It is called from the TIMER task, which is the highest-priority task in the system (and thus, will be the first to execute after start-up). INIT's role in life is to call FQ0GO (X) to initialize the FORTRAN I/O system, send one message to each of the lock exchanges (Y), and initialize the operator alarm panel (Z). The call to FQ0GO is a requirement for an RMX/80 system in which any FORTRAN-80 code that makes use of the iSBC 801 package is to be executed. The call must be made prior to the execution of any FORTRAN-80 I/O or math instructions. Also, the call to FQ0GO should only be made once.

DATE	TIME	PH	VOLUME	TEMP	DISSOLVED OXYGEN	TOTAL CARBON	ORGANIC CARBON
			(CU.M)	(C)	(MG/ML)	(MG/ML)	(MG/ML)
9/19/78	8:15:00	6.1	2614.32	25.4000	12.3400	76.9800	34.8870
9/19/78	8:10:00	6.2	2614.00	25.4000	12.5400	86.6340	40.4933

SUSPENDED SOLIDS	PHOSPHATE CONC	INFLUENT FLOW	EFFLUENT FLOW	TURBID	AIR DIS	MIX	INF
(MG/ML)	(MG/ML)	(MG/ML)	(MG/ML)				
16.8907	26.9800	112.890	0.000	74.56	ON	OFF	ON
19.3943	61.4300	119.340	0.000	86.43	ON	OFF	ON

Figure 19. Sample Output

SYSTEM GENERATION

Configuration Module

Now that all of the code for the individual tasks is written, it is time to generate the tables that give the RMX/80 Executive the information it needs to configure all of the tasks and exchanges. Assembly language macros are included in the RMX/80 product to help make building the configuration module a little easier. After the counters have been initialized, the STD macro is invoked several times to define Static Task Descriptors for the tasks in the system. Of special note are the last two entries (AA). Any task that uses the FORTRAN I/O system must allocate approximately 800 bytes of stack. This extra stack space is needed to save information on the

current I/O operations. Also, any task performing floating point calculations with the software package needs to append an extra 18 bytes to its Task Descriptor as a workspace area. If the iSBC 310 drivers are used this need be only 13 bytes long. This last field is defined by passing a value of 13 or 18 to the optional parameter, TDXTRA, in the STD macro. The routines in the FORTRAN-80 Run-Time Package require one exchange, F00LOK, which is allocated using the XCH macro (BB), and added to the Initial Exchange Table by the PUBXCH macro (CC).

Controller Addressable Memory Module

The CAMMOD module (DD) allocates the blocks of memory needed by the RMX/80 Disk File System. Specific details on the contents of this module can be found in the RMX/80 User's Guide.

LINK

Figure 20 shows the ISIS-II LINK command used to bind all of the individual modules together with the RMX/80 libraries needed to implement this application. The FORTRAN-80 I/O interface routines are found in the library F80RMX.LIB which is part of the iSBC 801 package. the library FPSFTX.LIB contains the software floating point package. If it is desired to accelerate the execution of the mathematical operations in this system, the iSBC 310 board can be included and the library changed to FPHRDX.LIB for iSBC 80/20, 80/20-4, and 80/30 systems or FPHX10.LIB for iSBC 80/10 and 80/10A systems.

The RMX/80 extensions included are the Disk File System, the Analog Handlers, and the Minimal Terminal Handler.

LOCATE

After the Link has been finished, the command shown in Figure 21 is used to invoke the ISIS-II LOCATE program. The ORDER statement sets the proper order for all of the

different segments and common blocks. The common blocks themselves are allocated as fixed blocks of memory to make possible their shared usage by PL/M routines using the AT attribute. This mechanism is discussed in greater detail in AP-44, "How to use FORTRAN With Other Intel Languages".

VII. SUMMARY

The purpose of this application note has been to describe the design process used to decide what operating system support to use, what language to code programs in, what hardware to use and what type of I/O to use to solve a given application problem. The specific application examples presented have keyed on the use of the FORTRAN-80 language.

The lesson that has been learned is that proper design techniques result in the use of the right tool for every job. With a complete set of programming languages, each optimized for a specific use, a powerful real-time executive, and a complete line of flexible hardware products, complicated applications become easy to solve.

```
LINK :F0:RMX80.LIB(START), &
:F1:X2CFG.OBJ, &
:F1:RPTMOD.OBJ, &
:F1:FRIMOD.LIB, &
:F1:INITMD.OBJ, &
:F0:F80RMX.LIB, &
:F0:F00LOK.LIB, &
:F0:FPSFTX.LIB, &
:F0:SYSTEM.LIB, &
:F0:DFSDIR.LIB(DIRECTORY,DELETE,RENAME,SEEK), &
:F0:DIO80.LIB, &
:F0:DFSUNR.LIB, &
:F1:CAM.OBJ, &
:F1:PLMMOD.LIB, &
:F0:AIHDLR.LIB, &
:F0:AOHDLR.LIB, &
:F0:MTI80.LIB, &
:F0:WTO80.LIB, &
:F0:RMX80.LIB, &
:F0:UNRSLV.LIB, &
:F0:PLMR0.LIB TO :F1:SEWAGE.LK0 PRINT(:F1:SEWAGE.LNK) MAP
```

Figure 20. LINK Command for Sewage Treatment Example

```
LOCATE :F1:SEWAGE.LK0 PRINT(:F1:SEWAGE.LOC) MAP &
CODE(0) STACKSIZE(0) LINES SYMBOLS PUBLICS &
ORDER(CODE DATA /LSTREC/ /STATUS/ /SETPNT/ /MIN5/) /STATUS/(FFA5H) &
/SETPNT/(FFDEH) /MIN5/(FFEEH) /LSTREC/(FFA3H)
```

Figure 21. LOCATE Command for Sewage Treatment Example


```

0016 D3ED 78 OUT CSTS ;
0018 3EB6 79 MVI A,TIMCMD ;SEND COMMAND WORD
001A D3DF 80 OUT TIMCP ;
001C 3EE0 81 MVI A,LOW(BDFCTR) ;SEND LOW ORDER BYTE
001E D3DE 82 OUT CNTR2 ;OF COUNTER VALUE
0020 3E00 83 MVI A,HIGH(BDFCTR) ;SEND HIGH BYTE OF
0022 D3DE 84 OUT CNTR2 ;COUNTER VALUE
0024 3E92 85 MVI A,CMD55 ;8255 COMMAND WORD
0026 D3EB 86 OUT PR255 ;PROGRAM 8255
0028 CD0000 E 87 CALL MAINLN ;CALL FORTRAN ROUTINE
002B C30000 C 88 JMP START ;IF ROUTINE RETURNS START OVER

```

```

90 ;
91 ; BUFIN--FILLS BUFFER WITH INPUT FROM TERMINAL
92 ;
93 PUBLIC BUFIN

```

```

002E E5 94 PUSH H ;SAVE HL PAIR
002F F5 95 PUSH PSW ;SAVE PSW
0030 C5 96 PUSH B ;SAVE BC
0031 60 97 MOV H,B ;GET BUFFER POINTER TO HL
0032 69 98 MOV L,C ;
0033 220000 D 99 SHLD BUFPTR ;SAVE IT
0036 1600 100 MVI D,0 ;ZERO TO # CHARACTERS COUNTER
;NOTE: STRING LENGTH <=255
0038 D5 102 PUSH D ;SAVE COUNTERS
0039 CDE200 C 103 GETCHR: CALL CI ;GET CHARACTER
003C FE7F 104 CPI RUBOUT ;RUBOUT?
003E C24700 C 105 JNZ BUF05 ;NO,CONTINUE
0041 CD9000 C 107 CALL DLTCHR ;YES,DELETE LAST CHARACTER

```

```

LOC OBJ SEQ SOURCE STATEMENT
0044 C33900 C 108 JMP GETCHR ;GET NEW ONE
0047 FE18 109 BUF05: CPI C ;CONTROL-X?
0049 CAA200 C 111 JZ DLTIN ;YES,DELETE BUFFER
004C FE12 112 CPI C ;CONTROL-R?
004E CAF900 C 113 JZ PRIBUF ;YES,PRINT BUFFER
0051 1D 114 DCR E ;DECREMENT SPACE LEFT COUNTER
0052 C26300 C 115 JNZ BUF10 ;CONTINUE IF COUNTER > 0
0055 FE0D 116 CPI CR ;IF THIS END OF LINE ALL IS OK
0057 CA6300 C 117 JZ BUF10
005A 1C 118 INR E ;BRING COUNTER BACK ONE
005B 0E07 119 MVI C,BELL ;NOT OK, ECHO BELL
005D CDB400 C 120 CALL ECHO ;
0060 C33900 C 121 JMP GETCHR ;GET NEW CHARACTER
0063 4F 122 BUF10: MOV C,A ;MOVE CHARACTER TO C
0064 CDB400 C 124 CALL ECHO ;AND ECHO IT
0067 71 125 MOV M,C ;STORE IT IN BUFFER
0068 23 126 INX H ;INCREMENT BUFFER POINTER
0069 14 127 INR D ;INCREMENT # OF CHARS COUNTER
006A 3E0D 128 MVI A,CR ;IS IT A NEWLINE CHARACTER
006C B9 129 CMP C ;
006D C23900 C 130 JNZ GETCHR ;NO,CONTINUE FILLING
0070 D1 131 POP D ;YES,RETURN
0071 C1 132 POP B ;
0072 F1 133 POP PSW ;
0073 E1 134 POP H ;
0074 C9 135 RET ;RETURN
136 ;
137 ; BUFOUT ENTRY POINT
138 ;
139 ;
140 PUBLIC BUFOUT
0075 E5 141 BUFOUT: PUSH H ;SAVE HL REGISTER PAIR
0076 F5 142 PUSH PSW ;SAVE PSW
0077 60 143 MOV H,B ;GET STRING POINTER INTO HL
0078 69 144 MOV L,C ;
0079 CDCD00 C 145 OUTCHR: CALL CHKIO ;CHECK FOR PAUSE(CNTR-S)
007C 4E 146 MOV C,M ;GET CHARACTER
007D CDB400 C 147 CALL ECHO ;OUTPUT TO TERMINAL
0080 3E0D 148 MVI A,CR ;IS IT A <CR>?
0082 B9 149 CMP C ;
0083 CA8D00 C 150 JZ EXITLP ;YES,EXIT
0086 23 151 INX H ;INCREMENT POINTER
0087 1B 152 DCX D ;DECREMENT STRING COUNT
0088 7A 153 MOV A,D ;GET HI BYTE
0089 B3 154 ORA E ;AND WITH LO BYTE
008A C27900 C 155 JNZ OUTCHR ;IF STRING COUNT <> 0,CONTINUE
008D F1 156 POP PSW ;RESTORE PSW
008E E1 157 POP H ;RESTORE HL
008F C9 158 RET ;ALL THROUGH
159 ;
160 ; DLTCHR--DELETES LAST CHAR ENTERED INTO BUFFER
161 DLTCHR:
0090 15 162 DCR D ;DECREMENT # OF CHARS COUNTER

```

LOC	OBJ	SEQ	SOURCE STATEMENT
		218 ;	
		219 ;	
		220 ;	CI--ENTER CHARACTER FROM TERMINAL
00E2	DBED	221 CI:	IN CSTS ;GET STATUS BYTE
00E4	E672	222 ANI	RXRDY ;CHARACTER AVAILABLE
00E6	CAE200	223 JZ	CI ;NO, LOOP
00E9	DBEC	224 IN	CDATA ;READY,GET CHARACTER
00EB	E67F	225 ANI	07FH ;STRIP OFF PARITY
00ED	C9	226 RET	
		227 ;	
		228 ;	CO--OUTPUT CHARACTER IN C REGISTER TO TERMINAL
		229 ;	
00EE	DBED	230 CO:	IN CSTS ;GET STATUS BYTE
00F0	E601	231 ANI	TXRDY ;TRANSMITTER READY?
00F2	CAEE00	232 JZ	CO ;NO, LOOP
00F5	79	233 MOV	A,C ;YES,MOVE CHARACTER TO ACC.
00F6	D3EC	234 OUT	CDATA ;SEND TO TERMINAL
00F8	C9	235 RET	
		236 ;	
		237 ;	PRTBUF--PRINTS CURRENT BUFFER(CONTROL-R)
		238 ;	
		239 PRTBUF:	
00F9	0E0D	240 MVI	C,CR ;ECHO CRLF
00FB	CDBA00	241 CALL	ECHO ;
00FE	E5	242 PUSH	H ;SAVE CURRENT BUFFER POINTER

FORTRAN COMPILER

10/12/78 PAGE 1

8 DUMMY=0
9 CALL FQFSET(DUMMY,DUMMY)

```

C-- CHECK FOR INVALID CHARACTERS
21 IF (KEYLTR.GE.1) THEN
22   IF (KEYLTR.LE.1#A) THEN
C
C-- IF VALID CHARACTER JUMP TO PROPER HANDLING ROUTINE
C
C      A B C D E F G H I J K L M N
23 (K) C  GOTO (300,100,100,300,100,100,100,100,100,100,100,100,100,100,
      O P Q R S T U V W X Y Z
      C100,100,100,100,300,200,100,100,100,100,100,100) KEYLTR

```



```

3      CHARACTER IMAGE*80,LINBUF(80)*1
4      INTEGER ERRNUM*2,INDEX*2,CARRET*1
5      EQUIVALENCE (LINBUF,IMAGE)
6      COMMON /LINE/ LINBUF,INDEX,CARRET
7
8      10 WRITE(IMAGE,10) ERRNUM,CARRET
9      FORMAT('***ERROR*** #',I4,A)
10     CALL BUFOUT(IMAGE)
11     RETURN
12     END

```

ISIS-II FORTRAN-88 COMPILATION OF PROGRAM UNIT MATH
 OBJECT MODULE PLACED IN :F1:MATH.OBJ
 COMPILER INVOKED BY: PORT88 :F1:MATH.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)

```

1      (O) SUBROUTINE MATH
2      IMPLICIT LOGICAL (A-Z)
3      C-- IMPLEMENTS CALCULATOR MODE
4      C
5      INTEGER INDEX*2,CARRET*1,ERRFLG*2
6      CHARACTER LINBUF(80)*1,IMAGE*80,COMMND*1,SYMBOL*1
7      REAL OP1,OP2,RESULT
8      EQUIVALENCE (LINBUF,IMAGE)
9      COMMON /LINE/ LINBUF,INDEX,CARRET
10
11     C-- RESCAN KEYLETTER TO DETERMINE OPERATION
12     C
13     INDEX=INDEX-1
14     COMMND=LINBUF(INDEX)
15     INDEX=INDEX+1
16     C-- MOVE INDEX TO FIRST OPERAND
17     C
18     CALL DBLANK
19     C-- GET IT IN
20     C
21     CALL CONVRT(OP1)
22     C-- REPEAT FOR SECOND OPERAND
23     C
24     CALL DBLANK
25     CALL CONVRT(OP2)
26     C-- PERFORM OPERATION
27     C
28     IF(COMMND.EQ.'M') THEN
29       RESULT=OP1*OP2
30       SYMBOL='*'
31       GOTO 11
32     ENDIF
33
34     IF(COMMND.EQ.'D') THEN
35       RESULT=OP1/OP2
36       SYMBOL='/'
37       GOTO 11
38     ENDIF
39
40     IF(COMMND.EQ.'A') THEN
41       RESULT=OP1+OP2
42       SYMBOL='+'
43       GOTO 11
44     ENDIF
45
46     IF(COMMND.EQ.'S') THEN
47       RESULT=OP1-OP2
48       SYMBOL='-'
49       GOTO 11
50     ENDIF
51
52     C-- OUTPUT RESULTS
53     C
54     11 WRITE(IMAGE,12,IOSTAT=ERRFLG,ERR=999) OP1,SYMBOL,OP2,RESULT,
55        ICARRET
56     12 FORMAT(F18.5,1X,A,1X,F18.5,1X,'=',1X,F18.5,A)
57     CALL BUFOUT(IMAGE)

```



```

38      RETURN
      C
      C--- ERROR HANDLER
      C
39      999 CALL ERROR(ERRFLG)
40      RETURN
41      END

```

FORTRAN COMPILER

10/12/78 PAGE 1

ISIS-II FORTRAN-80 COMPILATION OF PROGRAM UNIT CONVRT
 OBJECT MODULE PLACED IN :F1:CONVRT.OBJ
 COMPILER INVOKED BY: FORT80 :F1:CONVRT.FRT DEBUG DATE(10/12/78) PAGESWIDTH(78)

```

1      (R) SUBROUTINE CONVRT(VALUE)
2      IMPLICIT LOGICAL(A-Z)
      C
      C--- INPUTS NEXT PARAMETER IN LINBUF
      C
3      INTEGER I*1,INDEX*2,TMPIND*1,CARRET*1,ERRFLG*2
4      REAL VALUE
5      CHARACTER LINBUF(80)*1,TMPBUF(20)*1,BUFFER*20,ENDLIN*1
6      EQUIVALENCE (TMPBUF,BUFFER), (ENDLIN,CARRET)
7      COMMON /LINE/ LINBUF,INDEX,CARRET
      C
      C--- INITIALIZE
      C
8      DO 21 I=1,19
9      TMPBUF(I)=' '
10     TMPBUF(20)=ENDLIN
11     TMPIND=1
      C
      C--- FILL BUFFER UNTIL COMMA OR ENDLINE ENCOUNTERED
      C
12     22 TMPBUF(TMPIND)=LINBUF(INDEX)
13     INDEX=INDEX+1
14     TMPIND=TMPIND+1
15     IF (LINBUF(INDEX).EQ.',') THEN
16     INDEX=INDEX+1
17     (S) GOTO 23
18     ENDIF
19     IF (LINBUF(INDEX).EQ.ENDLIN) GOTO 23
20     GOTO 22
      C
      C--- READ UNDER FORMAT CONTROL
      C
21     23 READ(BUFFER,24,Iostat=ERRFLG,ERR=999) VALUE
22     24 FORMAT(F19.5)
23     (T) RETURN
      C
      C--- ERROR HANDLER
      C
24     999 CALL ERROR(ERRFLG)
25     RETURN
26     END

```

ISIS-II FORTRAN-80 COMPILATION OF PROGRAM UNIT TRANST
 OBJECT MODULE PLACED IN :F1:TRANST.OBJ
 COMPILER INVOKED BY: FORT80 :F1:TRANST.FRT DEBUG DATE(10/12/78) PAGESWIDTH(78)

```

1      SUBROUTINE TRANST
2      IMPLICIT LOGICAL (A-Z)
      C
      C--- PERFORM TRANSITION TESTING
      C
3      REAL START,STOP,STEP,TOL,TEMP,VOLTAG,VCC(20),
4      ILOWLVL(20),LOTOHI(20),HILVL(20),HITOLO(20)
      C
4      INTEGER CARRET*1,ITEMP*2,TSTINP*2,SAMPLE*2,
5      ILSTSAM*2,DELTA*2,ERRFLG*2,PNTCNT*1,INDEX*2,I*1
      C
5      CHARACTER LINBUF(80)*1,IMAGE*80,TEST(20)*6
6      EQUIVALENCE (LINBUF,IMAGE)
7      COMMON /LINE/ LINBUF,INDEX,CARRET
      C
      C--- INITIALIZE
      C

```

```

8      DO 5, I=1,20
9      5  TEST(I)='PASSED'
10     TSTINP=0
11     PNTCNT=1
      C
      C-- SCAN COMMAND TAIL FOR PARAMETERS
      C
      C      VCC START      STOP      STEP      TOLERANCE
      C
12     CALL DBLANK
13     CALL CONVRT(START)
14     CALL DBLANK
15     CALL CONVRT(STOP)
16     CALL DBLANK
17     CALL CONVRT(STEP)
18     CALL DBLANK
19     CALL CONVRT(TOL)
      C
      C-- IF (STOP-START)/STEP YIELDS MORE THAN 20 STEPS
      C-- OUTPUT MESSAGE AND RETURN
      C
20     IF (IFIX((STOP-START)/STEP).GT.20) THEN
21       WRITE(IMAGE,10,IOSTAT=ERRFLG,ERR=999) CARRET
22       10  FORMAT('TOO MANY POINTS',A)
23       CALL BUFOUT(IMAGE)
24       RETURN
25       ENDIF
      C
      C-- GET TEMPERATURE AND LINEARIZE
      C
26     CALL ADCIN(ITEMP,0)
27     TEMP=98.63*ALOG(FLOAT(ITEMP))+13.56
      C
      C-- OUTPUT HEADER
      C
28     WRITE(IMAGE,20,IOSTAT=ERRFLG,ERR=999) TOL,TEMP,CARRET,CARRET
29     20  FORMAT('TRANSITION TEST  TOLERANCE=',F5.1,
30     '  % AMBIENT TEMPERATURE = ',F6.2,' DEGREES C',A,A)
31     CALL BUFOUT(IMAGE)
32     WRITE(IMAGE,30,IOSTAT=ERRFLG,ERR=999) CARRET,CARRET
33     30  FORMAT('      VCC  HIGH TRANS  LOW TRANS  HIGH  LOW  TEST',
34     'A,A)')
35     CALL BUFOUT(IMAGE)
      C
      C-- BEGIN TEST; OUTPUT STARTING VCC VALUE
      C
34     VOLTAC=START
35     40  VCC(PNTCNT)=VOLTAC
36     CALL DACOUT(IFIX(VOLTAC*409.6),0)
      C
      C-- OUTPUT ZERO VOLTS TO TEST INPUT
      C
37     CALL DACOUT(TSTINP,1)
      C
      C-- GET ONE SAMPLE
      C
38     CALL ADCIN(SAMPLE,1)
      C
      C-- MAKE IT THE LAST SAMPLE AND ALSO STORE IT
      C
39     LSTSAM=SAMPLE
40     LOWLVL(PNTCNT)=FLOAT(SAMPLE)*409.6
      C
      C-- BEGIN LOOP LOOKING FOR LOW TO HIGH TRANSITION
      C
41     50  TSTINP=TSTINP+1
42     CALL DACOUT(TSTINP,1)
      C
      C-- GET SAMPLE
      C
43     CALL ADCIN(SAMPLE,1)
44     DELTA=SAMPLE-LSTSAM
      C
      C-- SEE IF TRANSITION; DELTA .GT. 2.2 VOLTS
      C
45     IF (DELTA.LT.901) THEN
46       LSTSAM=SAMPLE
      C
      C-- NO TRANSITION; IF TSTINP NOW UP TO 5.5V AND NO TRANSITION
      C-- OUTPUT MESSAGE INDICATING DEAD PART AND RETURN
      C
47     IF (TSTINP.GE.2251) THEN

```



```

78      C      TEST(PNTCNT)='FAILED'
      C-- INCREMENT VCC AND IF NOT .GT. STOP CONTINUE
      C
79      90      VOLTAG=VOLTAG+STEP
80      IF (VOLTAG.LE.STOP) THEN
81      PNTCNT=PNTCNT+1
82      TSTINP=0
83      GOTO 40
84      ENDIF
      C
      C-- TEST COMPLETE; OUTPUT RESULTS
      C
85      DO 110,I=1,PNTCNT
86      WRITE(IMAGE,100,IOSTAT=ERRFLG,ERR=999) VCC(I),
100      ILOTOHI(I),HITOL0(I),HILVL(I),LOWLVL(I),TEST(I),CARRET
87      110      FORMAT(3X,F5.2,3X,F6.2,6X,F6.2,3X,F6.2,1X,F6.2,2X,6A,A)
88      CALL BUFOUT(IMAGE)
89      RETURN
      C
      C-- ERROR HANDLER
      C
90      999      CALL ERROR(ERRFLG)
91      RETURN
92      END

```

FORTRAN COMPILER

10/12/78 PAGE 1

ISIS-II FORTRAN-80 COMPILATION OF PROGRAM UNIT ADCIN
 OBJECT MODULE PLACED IN :F1:ADCIN.OBJ
 COMPILER INVOKED BY: FORT80 :F1:ADCIN.FRT DEBUG DATE(10/12/78) PAGESWIDTH(78)

```

1  (X) SUBROUTINE ADCIN(VALUE,CHAN) (Y)
      C
      C-- ROUTINE TO INPUT SINGLE VALUE FROM A/D CONVERTER CHANNEL
      C-- GIVEN AND RETURN IT IN VALUE FIELD.
      C
2      INTEGER*2 VALUE
3      INTEGER*1 CHAN
4      SINCLUDE(:F1:ADCDAC.DEC)
      C
      C-- DEFINITIONS OF TSBG 732 REGISTERS
      C
      C-- COMMAND STATUS REGISTER
      C
5      INTEGER*1 CMDSTS
      C
      C-- MUX ADDRESS REGISTER
      C
6      INTEGER*1 MUXADR
      C
      C-- LAST CHANNEL REGISTER
      C
7      INTEGER*1 LSTCHN
      C
      C-- CLEAR INTERRUPT COMMAND WORD
      C
8      INTEGER*1 CLRINT
      C
      C-- ADC DATA REGISTER
      C
9      INTEGER*2 ADCDAT
      C
      C-- DAC 0 DATA REGISTER
      C
10     INTEGER*2 DAC0
      C
      C-- DAC 1 DATA REGISTER
      C
11     INTEGER*2 DAC1
      C
      C-- SET UP COMMON BLOCK
      C
12     COMMON /ADC/ CMDSTS,MUXADR,LSTCHN,CLRINT,ADCDAT,DAC0,DAC1
      C
      C-- SET UP CHANNEL ADDRESS
      C
13     MUXADR=CHAN
      C
      C-- START CONVERSION
      C
14     CMDSTS=#1H
      C
      C--

```

```

C
C-- WAIT FOR END OF CONVERSION
C
15 1 IF((CMDSTS.AND.#80H).NE.#80H) GOTO 1
C
C-- GET DATA IN
C
16 VALUE=ADCDAT
C
C-- RIGHT JUSTIFY AND CONVERT TO COUNTS
C
17 VALUE=VALUE/16
18 IF(VALUE.LT.0) VALUE=VALUE+4096+1
19 RETURN
20 END

```

ISIS-II FORTRAN-80 COMPILATION OF PROGRAM UNIT DACOUT
 OBJECT MODULE PLACED IN :F1:DACOUT.OBJ
 COMPILER INVOKED BY: FORT80 :F1:DACOUT.FRT DEBUG DATE(10/12/78) PAGES(78)

```

C
C-- SUBROUTINE DACOUT(VALUE,CHAN)
C
C-- OUTPUTS VALUE TO D/A CONVERTER
C
2 1 INTEGER*2 VALUE,CHAN
3 $INCLUDE(:F1:ADCDAC.DEC)
C
C-- DEFINITIONS OF ISBC 732 REGISTERS
C
C-- COMMAND STATUS REGISTER
C
4 1 INTEGER*1 CMDSTS
C
C-- MUX ADDRESS REGISTER
C
5 1 INTEGER*1 MUXADR
C
C-- LAST CHANNEL REGISTER
C
6 1 INTEGER*1 LSTCHN
C
C-- CLEAR INTERRUPT COMMAND WORD
C
7 1 INTEGER*1 CLRINT
C
C-- ADC DATA REGISTER
C
8 1 INTEGER*2 ADCDAT
C
C-- DAC 0 DATA REGISTER
C
9 1 INTEGER*2 DAC0
C
C-- DAC 1 DATA REGISTER
C
10 1 INTEGER*2 DAC1
C
C-- SET UP COMMON BLOCK
C
11 1 COMMON /ADC/ CMDSTS,MUXADR,LSTCHN,CLRINT,ADCDAT,DAC0,DAC1
C
C-- OUTPUT VALUE TO PROPER CHANNEL
C-- AFTER SHIFTING INTO HIGH ORDER 12 BITS
C
12 IF(VALUE.LT.0) VALUE=VALUE+4096+1
13 VALUE=VALUE*16
14 IF(CHAN.EQ.0) DAC0=VALUE
15 IF(CHAN.EQ.1) DAC1=VALUE
16 RETURN
17 END

```


APPENDIX B CODE LISTINGS

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE SEMAPHORES
 OBJECT MODULE PLACED IN :F1:SEMOD.OBJ
 COMPILER INVOKED BY: plm80 :F1:SEMOD.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)

```

1      SEMAPHORES:
      DO;

      /*****
      Contains LOCK and UNLOCK procedures for
      manipulating semaphores. Called by FORTRAN
      routines with one parameter; the address of
      the index of the semaphore to be operated on.
      *****/

      $nolist

17 1    DECLARE (stslok,setlok,dsklok) (10) BYTE PUBLIC;
18 1    DECLARE semaphore(3) ADDRESS PUBLIC DATA(
      .stslok,
      .setlok,
      .dsklok);
19 1    DECLARE token (3) STRUCTURE(
      link ADDRESS,
      length ADDRESS,
      type ADDRESS) PUBLIC;

20 1    LOCK: PROCEDURE(sema$number$ptr) REENTRANT PUBLIC;

21 2    DECLARE sema$number$ptr ADDRESS;
22 2    DECLARE sema$number BASED sema$number$ptr BYTE;
23 2    DECLARE msg$ptr ADDRESS;

24 2    msg$ptr=RQWAIT(semaphore(sema$number),0);
25 2    RETURN;
26 2    (A) END;

27 1    UNLOCK: PROCEDURE(sema$number$ptr) REENTRANT PUBLIC;

28 2    DECLARE sema$number$ptr ADDRESS;
29 2    DECLARE sema$number BASED sema$number$ptr BYTE;

30 2    CALL RQSEND(semaphore(sema$number),.token(sema$number));
31 2    (B) RETURN;
32 2    END;
33 1    END SEMAPHORES;

```

ISIS-II FORTRAN-80 COMPILATION OF PROGRAM UNIT STSINP
 OBJECT MODULE PLACED IN :F1:STSMOD.OBJ
 COMPILER INVOKED BY: FORT80 :F1:STSMOD.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)

```

1    (C) SUBROUTINE STSINP
2    IMPLICIT LOGICAL (A-Z)
      C
      C-- TASK CODE FOR STATUS INPUT TASK. UPDATES STATUS COMMON
      C-- BLOCK WITH ANALOG AND DIGITAL DATA VALUES. ALSO DOES
      C-- ANALOG COUNT TO ENGINEERING UNIT CONVERSIONS.
      C
3    CHARACTER SMPLEF*22,CLOCK*12
4    INTEGER*2 SAMPLES(11),DUMMY
5    REAL ANDATA(11)
6    EQUIVALENCE (SMPLEF,SAMPLES)
7    INTEGER*1 DIGDAT,I
8    COMMON /STATUS/ ANDATA,DIGDAT,CLOCK
      C
      C-- INITIALIZE FLOATING POINT LIBRARIES
      C
9    (D) DUMMY=0
10   CALL FQFSET(DUMMY,DUMMY)
      C
      C-- CALL INITIALIZATION ROUTINE
      C
11  (E) CALL INITIO
      C
      C-- CALL ROUTINE TO INPUT SAMPLES
      C
12  (F) 10 CALL SMPLIN(SMPLEF)
      C
      C-- SHIFT SAMPLES TO RIGHT JUSTIFY
      C

```



```

27 2      END;
28 1      TIMER5: PROCEDURE PUBLIC;
29 2          min$Sup=0;

          /* enter task loop */

30 2          DO WHILE 1;
31 3              time$Out=msg$ptr=RQWAIT(.min$Sup, five$minute$delay$Count);
32 3              min$Sup=time$Sup;
33 3              END; /* of do while 1 */
34 2      END; /* of procedure */
35 1      END; /* of module */

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE REPORT
OBJECT MODULE PLACED IN : F1:RPTMOD.OBJ
COMPILER INVOKED BY: plm80 :F1:RPTMOD.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)

1      REPORT:
      DO;

      /******
      This module contains the code for the REPORT
      task that prints formatted reports of system
      status upon command. Commands come in from
      PRTREQ exchange with type=100 for today's
      status report and type = 101 for yesterday's
      status report. PRINT is the FORTRAN routine
      that does the actual work.
      *****/

      $nolist

21 1      PRINT: PROCEDURE (file$ptr, name$size, request$type) EXTERNAL;
22 2          DECLARE (file$ptr, name$size) ADDRESS;
23 2          DECLARE request$type BYTE;
24 2      END PRINT;

25 1      FQFSET: PROCEDURE (A, ERRH) EXTERNAL;
26 2          DECLARE (A, ERRH) ADDRESS;
27 2      END FQFSET;

28 1      DECLARE prt$req (10) BYTE PUBLIC;

29 1      REPORT: PROCEDURE PUBLIC;

30 2          DECLARE today$type LITERALLY '100';
31 2          DECLARE yesterday$type LITERALLY '101';
32 2          DECLARE (ptr, dummy) ADDRESS;
33 2          DECLARE msg BASED ptr STRUCTURE(
              link ADDRESS,
              length ADDRESS,
              type BYTE,
              home$exchange ADDRESS,
              response$exchange ADDRESS);
34 2          DECLARE today$file$name (*) BYTE DATA('D0:TODAYS.RPT');
35 2          DECLARE ystday$file$name (*) BYTE DATA('D0:YSTDAY.RPT');

          /* initialize math handler */

36 2          dummy=0;
37 2          CALL FQFSET(.dummy, .dummy);

          /* enter task loop */

38 2      DO WHILE 1;
39 3          ptr=RQWAIT(.prt$req, 0);

          IF msg.type=today$type THEN
40 3              CALL print(.today$file$name, SIZE(today$file$name), .msg.t
41 3              - type);
42 3          ELSE IF msg.type=yesterday$type THEN
43 3              CALL print(.ystday$file$name, size(ystday$file$name), .msg
44 3              - .type);
          CALL RQSEND(msg.response$exchange, ptr);
45 3          END; /* of do while */
46 2      END; /* of task */
47 1      END REPORT;

```

ISIS-II FORTRAN-80 COMPILATION OF PROGRAM UNIT HEADER
 OBJECT MODULE PLACED IN : F1:PRNTMD.OBJ
 COMPILER INVOKED BY: FORT80 : F1:PRNTMD.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)

```

1      SUBROUTINE HEADER
C
C-- CALLED BY PRINT TO OUTPUT REPORT HEADER
C
2      WRITE(6,200)
3      200  FORMAT(' DATE      TIME PH      VOLUME      TEMP      DISSOLVED
- '
1'TOTAL ORGANIC SUSPENDED PHOSPHATE INFLUENT EFFLUENT ',
2'TURBID AIR DIS MIX INF')
4      WRITE(6,201)
5      201  FORMAT(44X,'OXYGEN      CARBON      CARBON      SOLIDS      CONC',6X,
1'FLOW
FLOW')
6      WRITE(6,202)
7      202  FORMAT(24X,'(CU.M)      (C)      (MG/ML)      (MG/ML) (MG/ML)
- '
1'(MG/ML)      (MG/ML)      (MG/ML) (MG/ML)
8      RETURN
9      END

```

ISIS-II FORTRAN-80 COMPILATION OF PROGRAM UNIT PRINT
 OBJECT MODULE PLACED IN : F1:PRNTMD.OBJ
 COMPILER INVOKED BY: FORT80 : F1:PRNTMD.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)

```

C
C-- SUBROUTINE PRINT CALLED BY REPORT TO GENERATE FORMATTED
C-- REPORTS. PRINTS EITHER TODAY'S FILE OR YESTERDAY'S
C-- DEPENDING ON FILNM INPUT VALUE.
C
1      SUBROUTINE PRINT(FILNM,TYPE)
2      IMPLICIT LOGICAL (A-Z)
3      CHARACTER*14 FILNM
4      INTEGER*2 ERRFLG,RECCNT,LSTREC
5      INTEGER*1 TYPE
6      INTEGER*1 INDEX
7      $INCLUDE(:F1:EQUIV.DEC)
8      CHARACTER BUFFER*57,PARAMS(57)*1
9      REAL PH,VOLUME,TEMP,DISOXY,TOTCAR,ORGCAR
10     REAL SUSSOL,PHOSFT,INFLOW,EFLFLO,TURBID
11     INTEGER*1 DIGDAT
12     INTEGER*2 MONTH,DAY,YEAR,HOURL,MINUTE,SECOND
13     EQUIVALENCE (PARAMS,BUFFER)
14     EQUIVALENCE (PARAMS,PH)
15     EQUIVALENCE (PARAMS(5),VOLUME)
16     EQUIVALENCE (PARAMS(9),TEMP)
17     EQUIVALENCE (PARAMS(13),DISOXY)
18     EQUIVALENCE (PARAMS(17),TOTCAR)
19     EQUIVALENCE (PARAMS(21),ORGCAR)
20     EQUIVALENCE (PARAMS(25),SUSSOL)
21     EQUIVALENCE (PARAMS(29),PHOSFT)
22     EQUIVALENCE (PARAMS(33),INFLOW)
23     EQUIVALENCE (PARAMS(37),EFLFLO)
24     EQUIVALENCE (PARAMS(41),TURBID)
25     EQUIVALENCE (PARAMS(45),DIGDAT)
26     EQUIVALENCE (PARAMS(46),MONTH)
27     EQUIVALENCE (PARAMS(48),DAY)
28     EQUIVALENCE (PARAMS(50),YEAR)
29     EQUIVALENCE (PARAMS(52),HOURL)
30     EQUIVALENCE (PARAMS(54),MINUTE)
31     EQUIVALENCE (PARAMS(56),SECOND)
32     CHARACTER*3 AIR,MIX,INFLNT,DISCHG
33     COMMON /LSTREC/ LSTREC
C
C-- INITIALIZE RECORD COUNT
C
34     RECCNT=1
C
C-- INITIALIZE INDEX
C
35     INDEX=1
C
C-- OUTPUT HEADER
C
36     CALL HEADER
C

```

```

C-- WAIT FOR FILE ACCESS IF TODAY'S FILE
C
37 1 IF (TYPE.EQ.100) CALL LOCK(2)
38 (U) OPEN(8,FILE=FILNM,STATUS='OLD',IOSTAT=ERRFLG,
100 1ERR=9000,ACCESS='DIRECT',RECL=57)
39 READ(8,REC=RECCNT,IOSTAT=ERRFLG,ERR=9100) BUFFER
40 RECCNT=RECCNT+1
41 IF ((DIGDAT.AND.#01H).EQ.#01H) THEN
42 AIR=' ON'
43 ELSE
44 AIR='OFF'
45 (V) ENDIF
46 IF ((DIGDAT.AND.#02H).EQ.#02H) THEN
47 MIX=' ON'
48 ELSE
49 MIX='OFF'
50 ENDIF
51 IF ((DIGDAT.AND.#04H).EQ.#04H) THEN
52 DISCHG=' ON'
53 ELSE
54 DISCHG='OFF'
55 ENDIF
56 IF ((DIGDAT.AND.#08H).EQ.#08H) THEN
57 INFLNT=' ON'
58 ELSE
59 INFLNT='OFF'
60 ENDIF
61 WRITE(6,101) MONTH,DAY,YEAR,HOUR,MINUTE,SECOND,
1PH,VOLUME,TEMP,DISOXY,TOTCAR,ORGCAR,SUSSOL,PHOSFT,
2INFLOW,EFLFLO,TURBID,AIR,DISCHG,MIX,INFLNT
62 101 FORMAT(I2,'/',I2,'/',I2,1X,I2,':',I2,':',I2,1X,F4.1,1X,F9.2,
1X,F9.4,1X,F9.4,1X,F8.3,1X,F8.3,1X,F9.4,1X,F9.4,1X,F8.3,1X,F8.3
-1X,F7.3,
1X,A3,1X,A3,1X,A3,1X,A3)
C
C-- CHECK FOR END OF FILE AND OTHER THINGS
C
63 INDEX=INDEX+1
64 IF (TYPE.EQ.100) THEN
65 IF (INDEX.LE.10) THEN
66 IF (RECCNT.LT.LSTREC) THEN
67 GOTO 10
68 ELSE
69 CLOSE(8,IOSTAT=ERRFLG,ERR=9200)
70 CALL UNLOCK(2)
71 (W) RETURN
72 ENDIF
73 ELSE
74 INDEX=1
75 CLOSE(8,IOSTAT=ERRFLG,ERR=9200)
76 CALL UNLOCK(2)
77 GOTO 1
78 ENDIF
79 ELSE
80 IF (RECCNT.LE.288) THEN
81 GOTO 10
82 ELSE
83 CLOSE(8,IOSTAT=ERRFLG,ERR=9200)
84 RETURN
85 ENDIF
86 ENDIF
C
C-- ERROR HANDLERS
C
87 9000 WRITE(6,9001) ERRFLG
88 9001 FORMAT('OPEN ERROR IN PRINT; #',I4)
89 RETURN
90 9100 WRITE(6,9101) ERRFLG
91 9101 FORMAT('READ ERROR IN PRINT; #',I4)
92 RETURN
93 9200 WRITE(6,9201) ERRFLG
94 9201 FORMAT('CLOSE ERROR IN PRINT; #',I4)
95 RETURN
96 END

```

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INITMD
 OBJECT MODULE PLACED IN :F1:INITMD.OBJ
 COMPILER INVOKED BY: plm80 :F1:INITMD.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)

```

1      INITMD:
      DO;
      $nolist
16 1    FQ0GO: PROCEDURE EXTERNAL;
17 2    END FQ0GO;

18 1    DECLARE semaphore (3) ADDRESS EXTERNAL;
19 1    DECLARE token (3) STRUCTURE(
      msgShdr) EXTERNAL;

20 1    INIT: PROCEDURE PUBLIC;
21 2    DECLARE i BYTE;

22 2    (X) CALL FQ0GO;
      /* initialize semaphores */

23 2    (Y) DO i=0 TO 2;
24 3    CALL RQSEND(semaphore(i),.token(i));
25 3    END;
      /* PROGRAM THE 8255 */

26 2    OUTPUT(0EBH)=92H;
      /* TURN OFF ALL ALARMS */

27 2    (Z) OUTPUT(0EAH)=0;

28 2    RETURN;
29 2    END;
30 1    END INITMD;
  
```

ASM00.OV3 :F1:X2CFG.M80 DEBUG PAGEWIDTH(78)

ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0 X2CFG PAGE 1

LOC	OBJ	SEQ	SOURCE STATEMENT
		1	NAME X2CFG
		2	CSEG
		3	PUBLIC RQRATE
0000 2000		4	RQRATE: DW 32
		5	\$NOLIST
		360	\$LIST
		361	\$NOGEN
0000		362	NTASK SET 0
0000		363	NEXCH SET 0
0000		364	NDEV SET 0
0000		365	NCONT SET 0
		366	;
		367	;
		368	BUILD INITIAL TASK TABLE
		369	STD RQADBG,64,129,RQWAKE
		426	STD RQTHDI,36,112,RQOUTX
		483	STD RQPSDK,48,129,RQDSKX
		540	STD RQDIR,48,130,RQDIRX
		597	STD RQDEL,64,131,RQDELX
		654	STD RQPRNM,64,132,RQPRNM
		711	STD RQAIH,34,133,RQAIEX
		768	EXTRN RQHD1
		769	CONSTD CNTRL,RQHD1,80,CNSTK,81,CONTEX
		882	STD TIMER,64,20,0
		939	STD TIMUPD,64,140,0
		996	STD TIMER5,64,141,0
		1053	STD STSINP,64,142,0
		1110	STD CHANGE,64,143,0
		1167	STD REPORT,800,144,0,18
		1224	STD SCAN,800,144,0,18
		1281	;
		1282	;
		1283	ALLOCATE TASK DESCRIPTORS
		1284	GENTD
		1288	;
		1289	ALLOCATE EXCHANGES
		1290	;
		1291	XCH CONTEX
		1295	XCH FQ0LOK

```

1299      INTXCH  RQLSEX
1305      ;
1306      BUILD  INITIAL EXCHANGE TABLE
1307      ;
1308      XCHADR  RQDSKX
1315      XCHADR  RQDIRX
1322      XCHADR  RQRNMX
1329      XCHADR  RQDELX
1336      XCHADR  RQAIEX
1343      PUBXCH  CONTEX
1350      PUBXCH  RQLSEX
1357      PUBXCH  FQ0LOK
1364      XCHADR  RQINPX

```

```

LOC  OBJ      SEQ      SOURCE STATEMENT
1371      XCHADR  RQOUTX
1378      XCHADR  RQBUG
1385      XCHADR  RQWAKE
1392      XCHADR  RQALRM
1399      XCHADR  RQL6EX
1406      XCHADR  RQL7EX
1413      XCHADR  STSLOK
1420      XCHADR  SETLOK
1427      XCHADR  DSKLOK
1434      XCHADR  BMPTIM
1441      XCHADR  TIMPOL
1448      XCHADR  PRTREQ
1455      XCHADR  CHRESP
1462      XCHADR  ANRESP
1469      XCHADR  MINSEX
1476      XCHADR  TIMEEX
1483      XCHADR  RDRESP
1490      ;
1491      BUILD  CREATE TABLE
1492      ;
1493      CRTAB
1500      ;
1501      BUILD  DEVICE CONFIGURATION TABLE
1502      ;
1503      DCT      D0,0,0,0
1504      DCT      D1,0,0,1
1505      ;
1506      BUILD  CONTROLLER SPECIFICATION TABLE
1507      ;
1508      CST      0,80H,5,RQLSEX,CONTEX
1509      ;
1510      BUILD  BUFFER ALLOCATION BLOCK
1511      ;
1512      BAB      3,BUFFPOL
1513      END

```

PL/M-80 COMPILER

10/12/78 PAGE 1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE CAMMOD

OBJECT MODULE PLACED IN: F1:CAM.OBJ

COMPILER INVOKED BY: plm80 :F1:CAM.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)

```

1      CAMMOD: DO;
      /* CONTROLLER TASK STACK */
2      1      DECLARE CANSSTK (80) BYTE PUBLIC;
      /* DFS INTERNAL BUFFER SPACE */
3      1      DECLARE RQDBUF (700) BYTE PUBLIC;
      /* DFS STATIC BUFFER POOL */
4      1      DECLARE BUF$POL (1200) BYTE PUBLIC;
5      1      END CAMMOD;

```


July 1980

2-75	INTRODUCTION
2-75	OVERVIEW
2-78	INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM
2-79	Architecture
2-79	Tasks
2-79	Job and Free Space Management
2-80	Segments
2-81	Communication & Synchronization
2-81	Interrupt Management
2-81	Error Management
2-81	Asynchronous I/O
2-82	Synchronous I/O
2-82	Loaders
2-83	File Management
2-83	Human Interface Subsystem
2-83	Debugging Subsystem
2-83	Configuration and Initialization
2-84	DESIGN METHODOLOGY
2-84	Application Example 1
2-84	System Requirements
2-84	Hardware Requirements
2-85	System Design
2-87	Application Example 2
2-87	Overview of Device Driver Construction
2-89	Design of an iSBC 284™ Driver
2-90	CODE EXAMPLES
2-97	APPENDIX A — Code Listings
2-121	APPENDIX B — Configuration Listings/Worksheet

Using the iRMX 86™ Operating System

Steve Verleye
OEM Microcomputer Systems Applications

Using the iRMX 86™ Operating System

Contents

INTRODUCTION	2-75
OVERVIEW	2-75
INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM	2-78
Architecture	2-79
Tasks	2-79
Job and Free Space Management	2-79
Segments	2-80
Communication & Synchronization	2-81
Interrupt Management	2-81
Error Management	2-81
Asynchronous I/O	2-81
Synchronous I/O	2-82
Loaders	2-82
File Management	2-83
Human Interface Subsystem	2-83
Debugging Subsystem	2-83
Configuration and Initialization	2-83
DESIGN METHODOLOGY	2-84
Application Example 1	2-84
System Requirements	2-84
Hardware Requirements	2-84
System Design	2-85
Application Example 2	2-87
Overview of Device Driver Construction ..	2-87
Design of an iSBC 534™ Driver	2-89
CODE EXAMPLES	2-90
APPENDIX A — Code Listings	2-97
APPENDIX B — Configuration Listings/Worksheets	2-121

INTRODUCTION

Companies seeking to develop microcomputer applications are faced with two significant problems. First, applications are growing more and more sophisticated. With competition always present, products are continually being enhanced with new features. This burdens the underlying computer system by increasing both the complexity of the software and the number of events and functions that must be handled by the system.

The second problem is a management problem. These newer and more sophisticated application systems must be developed quickly in order to hit shrinking market windows. Also, they must be developed with lower manpower costs to be feasible in an engineering community struck by insufficient technical personnel and skyrocketing software development costs.

These are the needs addressed by the iRMX 86™ Operating System. The two goals in the development of this product have been power/flexibility to meet the needs of increasingly complex application systems, and ease of understanding and use, to boost the productivity of available engineering resources. Users of Intel's line of iSBC 86™ Single Board Computers or custom-designed 8086-based boards can now obtain the same benefits from Intel supplied system software as they can from Intel supplied system hardware.

The reader of this application note is provided with information in four subject areas.

- The requirements of operating systems are discussed along with traditional solutions.
- The iRMX 86 Operating System is introduced and its features are discussed in relation to the requirements studied earlier.
- System design using the iRMX 86 Operating System is studied using example solutions.
- Code for two example systems is examined to learn the details of system implementation.

Some of the topics in this note may not be of interest to all readers. For example, an experienced real-time programmer may not need to read the entire overview of real-time systems. For those who want to brush up on a few topics, the overview is organized to allow the reader to focus attention on areas of specific interest.

Throughout this application note, various terms and concepts are introduced and discussed. If further information on any of these topics is desired, the references listed in the front of this note should be used.

OVERVIEW

This overview is provided to investigate both the problems encountered in the design of applications software and also the classical solutions to these problems.

Multitasking

A real-time system is defined to be a system that reacts to events occurring external to the computer and which monitors or controls these events as they occur (or in "real-time"). The converse of a real-time system is known as a batch system where the outcome of a program does not depend on when it is run (for example, a payroll program).

Two other characteristics typically encountered in a real-time system are asynchronous event occurrences and concurrent activity. The first characteristic is caused by events occurring randomly rather than at scheduled intervals. The second characteristic, concurrent activity, takes place when two or more events occur nearly at the same time, requiring simultaneous activity.

One method of dealing with the requirements of a real-time system would be to write a program that knows what events could potentially occur (for example, an interrupt occurrence, a real-time clock counting down to zero, a byte in memory being modified by another program). This program could then execute a large loop checking for the occurrence of these events.

There are several problems with this approach. While processing one event which has occurred, the program is not responsive to other events. Also, the programmer has no way of prioritizing the importance of the various events. From a maintenance standpoint, this program is complex and difficult to enhance or modify.

The traditional solution to these problems is a technique called multitasking. Essentially, this involves writing many small routines instead of one large one. Each of these routines (tasks) can process events independent of the other tasks in the system. In addition, a priority can be assigned each task so that the operating system can decide as to which task is the most important when more than one task requests control of the CPU.

The support for multitasking involves a scheduler which is part of the service provided by the operating system. The scheduler allows each task to execute its program as if it has sole control of the CPU, ensuring that all tasks desiring CPU time are serviced according to the priority associated with each task.

From the standpoint of system design, multitasking has many desirable qualities. Large and potentially complex application programs can be decomposed into smaller more manageable units. This makes feasible the use of programmer teams to implement the application. Perhaps even more importantly, the potentially overwhelming problems surrounding concurrent execution and interrupt handling become transparent to the application programmer. Also, multitasking makes the modification of existing tasks and the addition of new ones become a manageable objective since the interaction between tasks is minimized.

Interrupt Handling

A common event in a real-time system is the occurrence of an interrupt. Because this event is so common, an important feature of a real-time operating system is its interrupt processing capabilities.

From the standpoint of application software, interrupt handling can be cumbersome. The currently running task must be preempted, various hardware devices must be manipulated and perhaps a hardware interrupt controller must be dealt with.

A real-time operating system can abstract the occurrence of an interrupt into something more consistent with the way other events are handled. A task can simply inform the scheduler that it does not require any CPU time until an interrupt occurs. The relative priority of different interrupts can also be handled in the same manner as the priority of multiple tasks are handled. Thus, the application programmer need only deal with the actual processing related to interrupt occurrence.

Reliability

Reliability is a keyword in all real-time systems. In this type of system, reliability does not refer to mean time between failure. In fact, the software in a real-time application typically cannot be *allowed* to fail. The difficulty imposed on the software by the environment comes from the near infinite number of permutations that can occur. A system that appears to be fully debugged can fail in the field because of a combination of simultaneous events that never occurred before.

The only means to avoid failure in these instances is through the use of a consistent, well-thought-out model for handling events. Any special-cased solution is subject to failure when the special cases that were designed for are violated in the real world.

Error handling can also add reliability to an application system. When the application software is

unable to anticipate the outcome of certain conditions, or the software has undiscovered bugs, it is vital for the operating system to gracefully handle the situation and allow for further processing to continue as best as possible.

I/O Handling

Many applications for 16-bit microcomputers require a variety of I/O devices. The support for I/O operations on these devices is typically provided by the operating system. Both sequential access and random access devices are typically encountered and, in addition, flexibility in handling I/O requests and acknowledgements is important.

The flexibility necessary typically involves the scheduling of a task's execution after an I/O request has been made. The greatest flexibility can be obtained by an *asynchronous* I/O system. In this system, a task makes an I/O request by calling the operating system. Once the processing of the request has begun, control is returned to the calling task.

In this manner, the task can continue executing its program while the I/O operation is progressing. When the results of the operation are desired, the task can call the operating system again to wait for the completion of the previous I/O request.

The second type of I/O support is less flexible but also easier to use. An operating system that supports *synchronous* I/O allows a task to make a single operating system call to make an I/O request. Once control is returned to the calling task, the I/O operation is complete and the results are immediately available. This type of I/O support sometimes takes advantage of a technique known as *autobuffering* to regain some of the performance advantage of the overlapped I/O found in the asynchronous system.

Debug Support

The inherent characteristics of the real-time environment sometimes make it difficult to debug new software. If the simultaneous occurrence of two events causes a bug in the software, detection may be difficult because the next time the system is run the error is not reproduced. Also, because of the fact that the software is broken down into many independent tasks, the interaction may be difficult to track using standard debugging techniques.

The solution to these problems is a piece of software called the system debugger. The debugger typically has three characteristics.

- 1) It is designed to interact with the operating system and therefore has intimate knowledge of code, data structures and system objects.
- 2) Since the debugger is just another task in the system, it does not affect the operation of the other tasks that are running.
- 3) Through the use of sophisticated breakpointing facilities, the debugger allows the designer to track the tasks in the system, investigate their interaction with other tasks and selectively stop one or more tasks without stopping the entire system.

Multiprogramming

In some application systems, there arises the requirement to run several "applications" on the computer at the same time. This may be due to the desire to squeeze more use out of the hardware or it may be due to some system design consideration. These separate "applications" (often termed jobs) share many system resources (especially the CPU) but at the same time they need to be protected as much as possible from other jobs. In essence, it should be possible to develop two jobs independently and then run them both on the same hardware without any interaction. If interaction is desired, the operating system should support some well-defined protocol for jobs to use to communicate.

Free Space Management

One of the most important resources in the computer system is the memory. In some applications, the amount of memory needed can be determined when the system is designed. In the more general case, the amount of memory needed by the system fluctuates. One solution to this management problem is to have available the amount needed in the worst possible case. A more flexible and economical solution is to dynamically allocate memory from a central pool upon demand and return it when possible. This service provides two tangible advantages. First, total memory needs are reduced. Second, this service allows for ease of use by the application programmer because there is no need to set aside blocks of memory and implement code to maintain information about current usage.

File Management

The ability to easily store and retrieve data stored on mass storage devices is a requirement in many application systems. Devices such as disks, tapes and bubble memories are used to store program code, data files and parameter tables. The operating system is called upon to store and retrieve the data and organize it such that application programs can easily find and manipulate the data when necessary.

Typically, this service is provided through the use of a file system. The mass storage device is partitioned into blocks and logical addresses are assigned to the blocks. Files are created to serve as directories where the names of other files can be cataloged and looked up.

In many systems, the directory structure can go many levels deep (see Figure 1). This provides several advantages. Directory searches can be done much faster if the general area where a file exists is known. Also, if several jobs are running at the same time, each can be given its own directory and therefore isolated from the others. Lastly, for human users, it is much easier to manage the information on the disk when some logical structure of files exists.

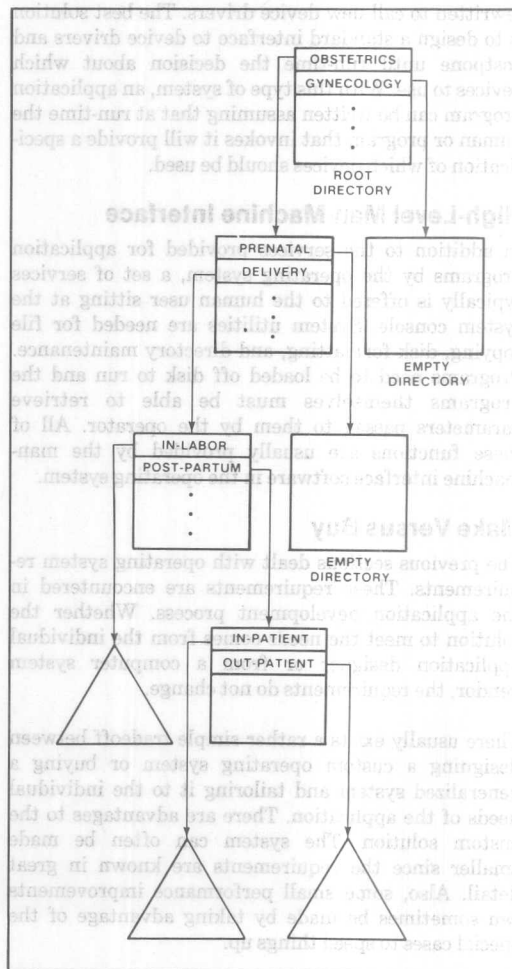


Figure 1. Hierarchical File System

Device Independence

One of the unfortunate characteristics of I/O devices is that they all tend to present different interfaces to the system software. When this is the case, the application programmer must become familiar with the unique characteristics of each device in order to communicate with it. One solution is to create an I/O driver which does the actual I/O. This driver can then be called by the application program whenever communication with the device is desired.

The problem with this solution is that the programmer must still know what type of device is being talked to since the I/O driver is specialized. If the system configuration changes, all of the software must be rewritten to call new device drivers. The best solution is to design a standard interface to device drivers and postpone until run-time the decision about which devices to use. With this type of system, an application program can be written assuming that at run-time the human or program that invokes it will provide a specification of which devices should be used.

High-Level Man-Machine Interface

In addition to the services provided for application programs by the operating system, a set of services typically is offered to the human user sitting at the system console. System utilities are needed for file copying, disk formatting, and directory maintenance. Programs need to be loaded off disk to run and the programs themselves must be able to retrieve parameters passed to them by the operator. All of these functions are usually provided by the man-machine interface software in the operating system.

Make Versus Buy

The previous sections dealt with operating system requirements. These requirements are encountered in the application development process. Whether the solution to meet the needs comes from the individual application designer or from a computer system vendor, the requirements do not change.

There usually exists a rather simple tradeoff between designing a custom operating system or buying a generalized system and tailoring it to the individual needs of the application. There are advantages to the custom solution. The system can often be made smaller since the requirements are known in great detail. Also, some small performance improvements can sometimes be made by taking advantage of the special cases to speed things up.

Buying an operating system from a computer system vendor offers five advantages.

- 1) Engineering resources are becoming scarce. The use of an operating system from a vendor allows attention to be focused on the application software.
- 2) The time taken to bring the product to market can be shortened, thereby gaining a competitive edge and generating early revenue.
- 3) Long-term maintenance costs can be reduced because the vendor supports the operating system software.
- 4) Personnel in all branches of the company can become familiar with one software architecture and apply this knowledge to a range of products. This applies not only to the design engineers, but also to quality assurance, customer engineers and system analysts.
- 5) The computer system vendor has knowledge of future technological advances coming in the product lines. For this reason, the operating system can be constructed so that applications software can be transported to future hardware without the need for expensive redesign.

In summary, the trade-offs are clear. An operating system from a computer system vendor is not the answer for every application. But in most cases, the most economical and safest bet is to take advantage of the expertise of the vendor for the system software and use engineering resources to more quickly solve the application problem.

INTRODUCTION TO THE iRMX 86™ OPERATING SYSTEM

The iRMX 86 Operating System meets the needs of real-time applications while simultaneously providing the full set of services normally found in a general-purpose operating system.

The overall picture of the iRMX 86 Operating System is shown in Figure 2. The iRMX 86 Nucleus provides

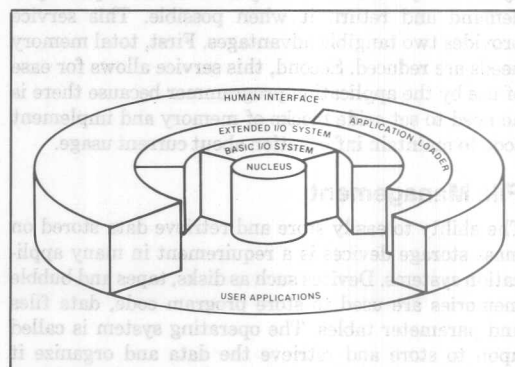


Figure 2. Layers of Support in the iRMX 86™ System

support for multitasking, multiprogramming, inter-task communication, interrupt handling and error checking. The Basic I/O System provides support for device independent and file format independent manipulation of data on I/O devices. The Extended I/O system provides synchronous I/O calls, automatic buffering, logical file name support and high-level job management. The application loader provides the ability to load code and data from mass storage devices into RAM memory. The Human Interface provides for a high-level man-machine interface as well as file utilities and parsing support for application programs.

The following sections deal in more detail with each of these iRMX 86 pieces. If more information is desired on the features discussed, please refer to the documents listed in the front of this application note.

Architecture

The iRMX 86 architecture is an object-oriented architecture. This means that the operating system is organized as a collection of building blocks that are manipulated by operators. The building blocks of the iRMX 86 system are called objects and are of several types. Some of the object types are tasks, jobs, mailboxes, semaphores and segments. These types are explained in subsequent sections of this application note.

This type of architecture has two major advantages. First, the system is easier to learn and use. The attributes of the various objects and the operations that can be performed on them are well defined and consistent. Once an object type is understood, all objects of that type are understood.

The second advantage to an object-oriented architecture is the ease with which the operating system can be tailored to the application. If there is no need for a given object in the application, all operators for that object are not included in the final configured system. On the other hand, if the application designer needs a more complex building block that is not in the basic system, he can define and use a new object type.

Table 1 lists all of the system calls in the iRMX 86 Nucleus. There are three groupings of system calls in this table.

- 1) The general system calls apply to all objects uniformly.
- 2) The first two system calls for each object are the create and delete calls. These calls simply create a new object and initialize its attributes or delete an existing object.

- 3) The remaining system calls are specific to the attributes of a particular object. With this organization in mind, the entire operation of the iRMX 86 nucleus can be glimpsed in a single table.

Tasks

Tasks are the active objects in the iRMX 86 architecture. Tasks execute program code and therefore are the only objects that can manipulate other objects. The attributes of a task include its program counter, stack, priority and dispatcher state.

Tasks compete with each other for CPU time and the iRMX 86 scheduler determines which task to run based upon priorities. The dispatcher states for an iRMX 86 task are shown in Figure 3. At any given point in time, the highest priority task that is ready to run has control of the CPU. Control is transferred to another task only when

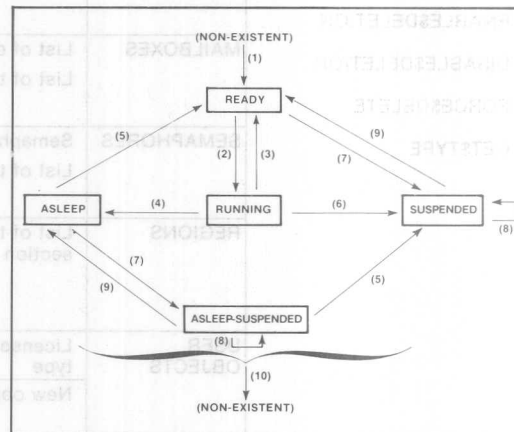


Figure 3. Task State Transition Diagram

- 1) the running task makes a request that cannot immediately be filled and is, therefore, moved to the asleep state,
- 2) an interrupt occurs causing a higher-priority task to become ready to run or
- 3) the running task causes a higher-priority asleep task to become ready by releasing some resource.

The suspended and asleep-suspended states are entered whenever the suspend system call is invoked for a particular task.

Job and Free Space Management

Support for multiprogramming is provided by the job object. A job provides the environment for tasks to execute their programs. All other objects needed for a particular application are contained within the job.

Table 1. Nucleus Object Management System Calls

System Calls for All Objects	O.S. Objects	Attributes	Object-Specific System Calls
	JOB	Tasks Memory pool Object directory Exception handler	CREATE\$JOB DELETE\$JOB SET\$POOL\$MIN GET\$POOL\$ATTRIB OFFSPRING
CATALOG\$OBJECT UNCATALOG\$OBJECT LOOKUP\$OBJECT	TASKS	Priority Stack Code State Exception handler	CREATE\$TASK DELETE\$TASK SUSPEND\$TASK RESUME\$TASK GET\$EXCEPTION\$HANDLER SET\$EXCEPTION\$HANDLER SLEEP GET\$TASK\$TOKENS GET\$PRIORITY SET\$PRIORITY
ENABLE\$DELETION DISABLE\$DELETION FORCE\$DELETE GET\$TYPE	SEGMENTS	Buffer with length	CREATE\$SEGMENT DELETE\$SEGMENT GET\$SIZE
	MAILBOXES	List of objects List of tasks waiting for objects	CREATE\$MAILBOX DELETE\$MAILBOX SEND\$MESSAGE RECEIVE\$MESSAGE
	SEMAPHORES	Semaphore unit value List of tasks waiting for units	CREATE\$SEMAPHORE DELETE\$SEMAPHORE RECEIVE\$UNITS SEND\$UNITS
	REGIONS	List of tasks waiting for critical section	CREATE\$REGION DELETE\$REGION RECEIVE\$CONTROL ACCEPT\$CONTROL SEND\$CONTROL
	USER OBJECTS	License rights to a given extension type New object template	CREATE\$EXTENSION DELETE\$EXTENSION CREATE\$COMPOSITE DELETE\$COMPOSITE INSPECT\$COMPOSITE ALTER\$COMPOSITE

A specific attribute of the job is a free memory pool from which blocks can be allocated only by tasks within the job. Also, the job contains an object directory which can be used by tasks to catalog objects under ASCII names so that other tasks, knowing the ASCII name, can look up the object and thereby gain addressability to it.

More than one job can co-exist in the computer system. Tasks within jobs can also create children jobs forming a hierarchical tree of jobs (see Figure 4). Each job in the system has its unique set of contained objects, its own memory pool and its own object directory.

Segments

A fundamental resource that tasks need is memory. Memory is allocated to tasks in the form of the

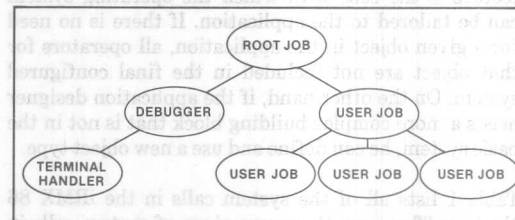


Figure 4. iRMX 86™ Job Tree Example

segment object. The segment is a block of contiguous memory. The attributes of a segment are its base address and size. A task needing memory requests a segment of whatever size it requires. The Nucleus attempts to create a segment from the memory pool given to the task's job when the job was created.

If there is not enough memory available, the Nucleus will try to get the needed memory from ancestors of the job.

Communication and Synchronization

In many cases it is necessary for two tasks to communicate in order to exchange data and commands. This is supported through the use of an object known as a mailbox. As its name implies, a mailbox is a holding place for objects. One task can send an object to a mailbox, causing the object to be queued there. Another task can later receive an object from the mailbox and thereby gain access to it (see Figure 5). If a task tries to receive an object from a mailbox and there are no objects there, the task can optionally be made to sleep for a specified time for an object to appear.

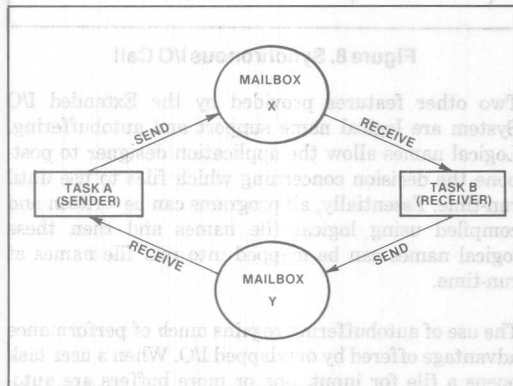


Figure 5. Intertask Communication via Mailboxes

Note that any object can be sent to a mailbox to be received by another task. Typically, the object sent is a segment which is a block of memory and can contain any commands or data. The term message is often used to describe the object during the time it is being sent through a mailbox.

In those cases where there is a requirement for synchronization between tasks but no data need be sent, a simpler more efficient mechanism exists. The semaphore object provides for the allocation of abstract entities called units. The primary attribute of the semaphore is an integer number. Tasks may send units to a semaphore thereby increasing the integer number or they can request units, thereby decreasing the number. If a task makes a request for more units than are available, it can optionally be made to sleep for a specified amount of time. This mechanism can be used for synchronization, resource allocation and mutual exclusion.

Interrupt Management

When an interrupt is sensed by the 8086 hardware, a user interrupt handler is executed. The interrupt handler can either perform all interrupt processing itself without making any iRMX 86 system calls, or it can signal an interrupt task allowing more general interrupt processing including calls to the operating system.

The operating system maps hardware interrupt priorities into the software priority scheme allowing the designer to specify what software functions are important enough to have some interrupt levels masked off during their execution. Although this mapping should always be kept in mind during design, the mechanics of dealing with interrupt control are handled by the operating system.

Error Management

One of the central themes in the design of the iRMX 86 operating system has been reliability. The results of these efforts are evident in two particular features of the architecture. Beyond the ease of understanding brought about by the symmetry of the system, the reliability of applications using the iRMX 86 software is increased.

The general case (as opposed to checking only for specific combinations of errors) has been designed for. Because of this, an unexpected combination of events or the simultaneous occurrence of interrupts will never catch the system by surprise.

In the event that errors do occur, the operating system is set to detect them. Virtually all parameters in calls to the operating system are checked for validity. Any inconsistency causes a jump to an error routine to handle the problem. Two types of errors can potentially occur and there are two ways of handling errors.

The first error type is the programmer error condition which comes about due to some mistake in the coding of a system call. The second type is an environmental condition which arises due to factors out of the control of the engineer (e.g. insufficient memory). Each of these error types can be handled in-line by checking a status code upon return from the call or can cause an error handling subroutine to be called by the system. The system designer can choose the desired method for the system, for a specific job, and even for individual tasks within a job.

Asynchronous I/O

Asynchronous I/O system calls are provided to support device independent I/O to any device in the

system. The type of I/O and the type of device are interrelated as shown in Figure 6. Every device driver in the I/O system is required to support a standard interface. In this manner, all devices look the same to higher level software. In the same manner, the individual file drivers, which provide the different types of file systems, all have a standard interface and call upon the various device drivers to perform I/O. These interface standards

- 1) provide for the device independence in the higher layers of the I/O system
- 2) make it easier for Intel to add future device drivers as new devices become available and
- 3) make it possible for iRMX 86 users to add their own drivers for custom I/O devices.

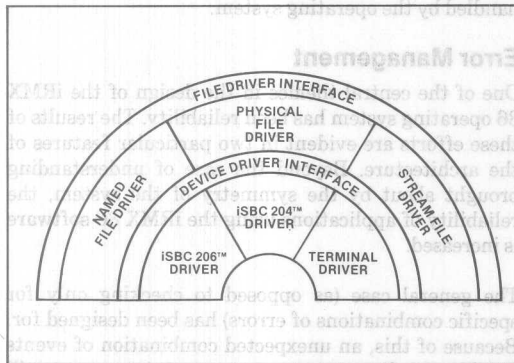


Figure 6. I/O System Structure

The iRMX 86 I/O system provides both asynchronous and synchronous system calls. The asynchronous I/O calls are faster, provide more flexibility in the selection of options and allow the program making the call to perform other functions while waiting for the I/O operation to complete.

The method by which the I/O system responds to the requestor is through the use of a mailbox. When any call is made to the asynchronous I/O system, one of the parameters indicates a mailbox where the caller expects to receive a segment containing the results of the operation (see Figure 7).

Synchronous I/O

The alternative to using the asynchronous I/O system is to use synchronous I/O system calls. As shown in Figure 8, the number of options available are fewer and the caller cannot continue execution until the entire I/O operation is completed but from an ease-of-use standpoint, the situation is much simplified.

```
Response$mailbox$token = RQ$create$
mailbox (0, @status);
CALL RQ$$read(connection$token,buf$ptr,
count,response$mailbox$token,@status);
IORS$token = RQ$receive$message
(response$mailbox$token,OFFFHH,
@resp$t,@status);
{check status}
Call RQ$delete$segment(IORS$token,
@status);
```

Figure 7. Asynchronous I/O Call

```
Call RQ$$$read(connection$token,buf$ptr,
count,@status);
{check status}
```

Figure 8. Synchronous I/O Call

Two other features provided by the Extended I/O System are logical name support and autobuffering. Logical names allow the application designer to postpone the decision concerning which files to use until run-time. Essentially, all programs can be written and compiled using logical file names and then these logical names can be mapped into real file names at run-time.

The use of autobuffering regains much of performance advantage offered by overlapped I/O. When a user task opens a file for input, one or more buffers are automatically created and filled with data from the file. Thus, when the user task makes an I/O request, the data may already be available in memory. A similar case exists for write requests in that the I/O system will buffer data to be written to a device, allowing the user task to continue on.

Loaders

The iRMX 86 application loader and bootstrap loader perform a variety of services for the user software. The following is a brief summary of the available features.

- 1) Systems can be boot loaded from mass storage devices at system reset. This saves not only ROM or EPROM memory, but also reduces field maintenance costs by allowing easy field updates.
- 2) Users can design their own SYSGEN procedure allowing tailoring of an application system to the individual installation.
- 3) Infrequently used programs can be brought in from mass storage when needed instead of using system memory unnecessarily.

File Management

There are three types of files supported by the iRMX 86 I/O system, named files, physical files and stream files. Named files are supported on devices possessing mass storage capability. Files in this system have ASCII pathnames and are cataloged in directories. Each device in the system contains a directory tree as shown previously in Figure 1. Access protection is provided through the use of access lists for each file. Each user or group of users in the system can be given different types of access to the file or can be denied access to it.

For devices that cannot support a named file structure (e.g. printers and terminals) the physical file driver is used. Devices in this category are treated strictly as data going into and/or out of the device. If it is desirable to treat a mass storage device strictly as a large mass of data, it can also be addressed through the physical file driver.

The third type of file is the stream file. This file type has no correlation with any physical device but rather uses system memory for temporary storage of data. An example of the usage of a stream file is a job that gets its input stream of data from a file. Depending on which time the job is run, this file might be a named file on disk, a terminal, or a stream file being written to by another job (see Figure 9).

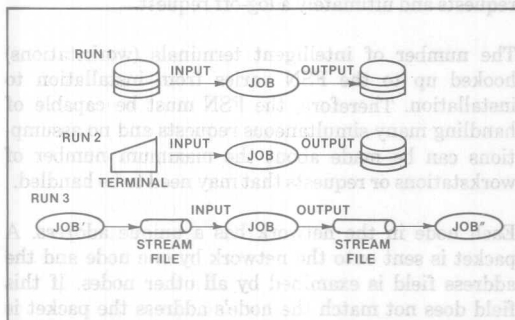


Figure 9. Stream File Example

Human Interface Subsystem

The highest level of support provided by the iRMX 86 Operating System is the Human Interface Subsystem. This piece of software provides two basic services. Programs can be invoked by typing the program name at the system console. The Human Interface will load the given program into memory, set it up as a job and start it running. The invoked program can then call upon the Human Interface routines to determine what parameters were passed to it as part of the operator input.

The Human Interface also contains a set of system utility routines which are used to copy files and disks, format disks, dynamically alter the system configuration and others.

Debugging Subsystem

The iRMX 86 Debugging Subsystem allows the designer to interact with the prototype system and isolate and correct program errors. Since the debugger is an object-oriented debugger and is aware of the internal structure of the operating system, it can provide detailed information concerning objects and can monitor mailboxes and semaphores providing a breakpoint facility as well as error detection.

Specifically, the iRMX 86 Debugging Subsystem provides six sets of functions:

- 1) Wake-up upon operator invocation. The operator types a control-D key to cause the debugger to wake up.
- 2) View system lists. The debugger can view lists of objects either globally or specifically for a given job. Also, lists of objects and tasks queued at mailboxes and semaphores can be seen.
- 3) Inspect objects. A detailed report on any object can be requested showing the current state of all relevant attributes.
- 4) Inspect and modify memory.
- 5) Breakpoint control. Any number of breakpoints can be set causing a single task to break on either execution of particular instructions or sends and receives of messages or units.
- 6) Error handling. The debugger can be set up to be the system default error handler thus catching system exceptions.

Configuration and Initialization

Once the application is designed and coded, the engineer needs a mechanism to inform the operating system of the software and hardware configuration. Essentially, this involves building tables of information using tools provided with the iRMX 86 product.

As shown earlier in Figure 4, the jobs in an iRMX 86 system form a hierarchical tree. The root in every job tree is known as the root job and is supplied as part of the iRMX 86 system. There are three important features of this job.

- 1) The root job has an object directory for cataloging and looking up objects. The special feature of this directory is that it is accessible by all tasks in the system since everyone can address the root job. For this reason the root object directory is useful for setting up inter-job communication paths.

- 2) The root job initially contains all free space in the system. Part of the system initialization code performs a memory scan to automatically determine the amount of free RAM in the system. This memory is put into the free space pool of the root job and parceled out as user jobs are created.
- 3) The root job contains only one task, the root task. This task scans the configuration tables generated by the user and creates the user-specified jobs.

Examples of configuration, initialization and the LINK 86 and LOC 86 operations needed to generate a system will be presented in the Code Examples section.

DESIGN METHODOLOGY

This section describes the design process involved in using the iRMX 86 system to solve application problems and presents two example solutions.

System design with the iRMX 86 Operating System should be viewed as a process starting with the highest level definition of system requirements and successively adding more detail until the end product is program code. This description sounds very much like the description of top-down design and, of course, it should. This methodology offers not only quicker designs, fewer design flaws and easier implementation, but also easier maintenance and enhancement.

In general, every iRMX 86 design progresses through the following steps:

- 1) Define system requirements.
- 2) Breakdown into highest level sub-functions (jobs).
- 3) Define job functions.
- 4) Determine inter-job command and data flow.
- 5) Break down each job into sub-functions.
- 6) Based upon requirements, assign tasks to perform job functions.
- 7) Determine inter-task command and data flow.
- 8) Write program code for each task.

Step 8 becomes the design process associated with the application programs themselves. The code for each task is essentially a sequential program that performs one of the functions of the computer system. Standard techniques for top-down design can therefore be used here to specify each module and its inputs and outputs as well as global and local data structures etc. The end product of this procedure is a modularized application system that should be easy to debug.

APPLICATION EXAMPLE 1

The first example presented here is based on the distributed local network diagrammed in Figure 10. Each

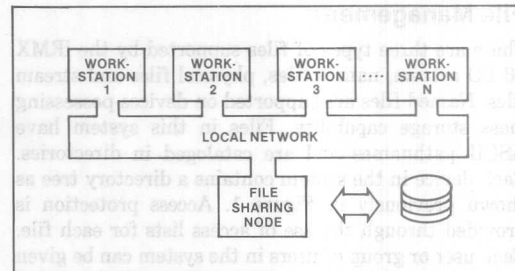


Figure 10. Block Diagram of Example System 1

workstation shown is an intelligent terminal having local data and program storage. The stations all use the File Sharing Node (FSN) for storage and retrieval of records in much the same way as the secretaries in an office would make use of a filing cabinet. The FSN maintains the files on a fixed disk device and responds to requests from the workstations for access to the data. The design to follow concentrates on the File Sharing Node.

System Requirements

Each intelligent terminal in the network has command processing software. When a file reference is made that cannot be satisfied by the local file system, a request is made to the File Sharing Node. This request consists of a log-on request followed by a string of I/O requests and ultimately a log-off request.

The number of intelligent terminals (workstations) hooked up to the FSN varies from installation to installation. Therefore, the FSN must be capable of handling many simultaneous requests and no assumptions can be made about the maximum number of workstations or requests that may need to be handled.

Each node in the network has a unique address. A packet is sent onto the network by one node and the address field is examined by all other nodes. If this field does not match the node's address the packet is ignored. If a match is found the packet is retrieved from the network.

Hardware Requirements

The three main hardware building blocks needed by this application are shown in Figure 11. The iSBC 86/12A Single Board Computer will communicate with the iSBC 544 Intelligent Communications Controller to establish and maintain communications with the network. The Intel 8085A on the iSBC 544 board will perform all of the address recognition, acknowledgements, packet retrieval and packet transmittal. The iSBC 206 Hard Disk Controller will be used to

create, maintain and access the data files which are at the heart of this application.

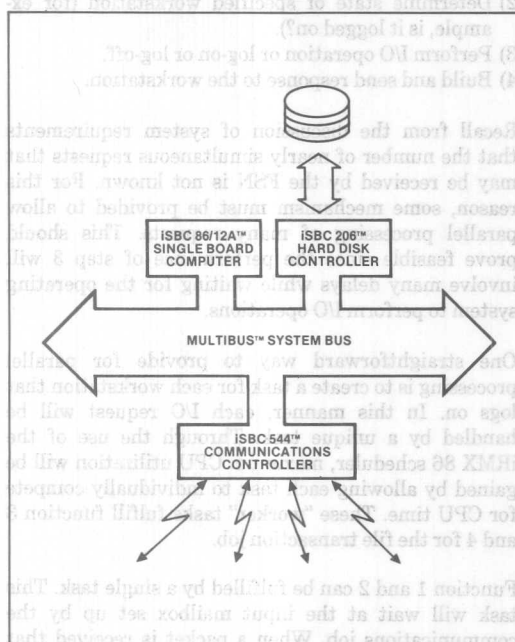


Figure 11. Hardware Block Diagram

System Design

The first step in the system design process is the breakdown of the system functions into one or several jobs. The reasons for doing this are system modularity and protection. With this type of design, each job can be designed separately, perhaps even by a different engineer or engineering team. The input and output requirements will be specified very tightly and the job will take on the appearance of a black box to other jobs in the system. If the job is enhanced or modified at a later date, the rest of the system can be left undisturbed providing that the input and output response remains the same.

The job object in the iRMX 86 operating system also affords a degree of software protection for the tasks and other objects contained within the job. Each job has a separate memory pool, a separate object directory and a separate identification to the I/O system.

The two primary groupings of functions in this application are those related to the network communications and those related to processing the file transaction request. A list of a possible split-up of system functions is shown in Figure 12.

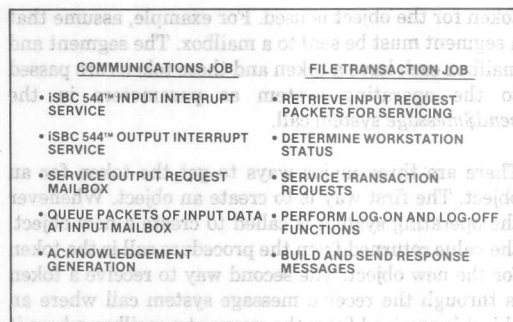


Figure 12. Function Split-up

The communication between the file transaction job and the communication job must fulfill two basic needs. The communication job will receive interrupts when packets addressed to the FSN are received. In order to remain attentive to new requests coming in, the communications job should have the capability to "spool" the requests off to the file transaction job. This buffering can be provided by using the mailbox object. Segments can be created to contain the packet request data and can then be sent to a mailbox where the file transaction job can receive and process them.

When the file transaction job must send a packet to a workstation, the requirement is seen for another queue of requests. Since the communications board can only put one packet at a time on the network, a mailbox should be provided to allow tasks in the file transaction job to send output request segments into the queue and then continue on (see Figure 13).

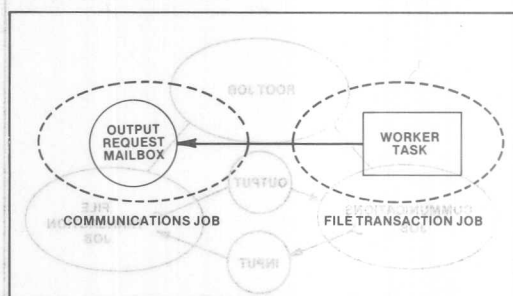


Figure 13. Output Mailbox Queue

Since tasks in both the file transaction job and the communications job must have access to these input and output mailboxes, some means must be set up to "broadcast" the identifier for these objects.

In the iRMX 86 system, each object has associated with it a 16-bit number called a token. Whenever an object is referenced in an operating system call, the

token for the object is used. For example, assume that a segment must be sent to a mailbox. The segment and mailbox each have a token and these tokens are passed to the operating system as parameters in the *send\$message* system call.

There are three major ways to get the token for an object. The first way is to create an object. Whenever the operating system is called to create a new object, the value returned from the procedure call is the token for the new object. The second way to receive a token is through the receive message system call where an object is received from the queue at a mailbox where it was sent by another task.

The third major mechanism for the receipt of a token is provided by the object directory concept. As mentioned previously, each job in the system has an object directory.

If a task in a job has the token for an object and wishes to let other tasks in other jobs have access to the object, the task can "catalog" the object in the object directory. The *catalog\$object* system call takes the token for an object and an ASCII name as parameters and creates an entry in the object directory. If another task knows the ASCII name for an object, it can obtain the token by performing a *lookup\$object* call.

The object directory mechanism will be used in this example to allow the communications job to "broadcast" the tokens for the input and output mailboxes. The jobs for this application are shown in Figure 14.

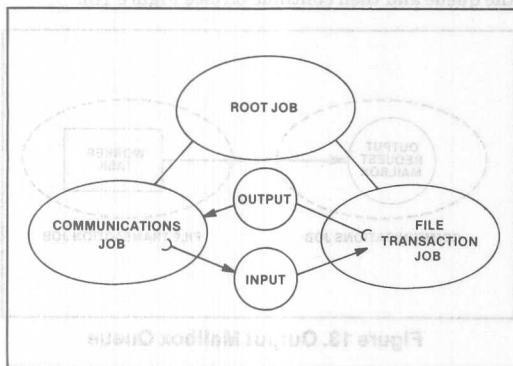


Figure 14. Job Structure

The next step of the design methodology calls for each job to be further divided into sub-functions. In this application note, only the file transaction job is studied.

In time sequence, the file transaction job will:

- 1) Retrieve input requests from the mailbox set up by the communication job.
- 2) Determine state of specified workstation (for example, is it logged on?).
- 3) Perform I/O operation or log-on or log-off.
- 4) Build and send response to the workstation.

Recall from the discussion of system requirements that the number of nearly simultaneous requests that may be received by the FSN is not known. For this reason, some mechanism must be provided to allow parallel processing of many requests. This should prove feasible since the performance of step 3 will involve many delays while waiting for the operating system to perform I/O operations.

One straightforward way to provide for parallel processing is to create a task for each workstation that logs on. In this manner, each I/O request will be handled by a unique task. Through the use of the iRMX 86 scheduler, maximum CPU utilization will be gained by allowing each task to individually compete for CPU time. These "worker" tasks fulfill function 3 and 4 for the file transaction job.

Function 1 and 2 can be fulfilled by a single task. This task will wait at the input mailbox set up by the communications job. When a packet is received that requests a log-on operation, the "listener" task will create a new "worker" task to handle the request. Figure 15 shows a picture of the design.

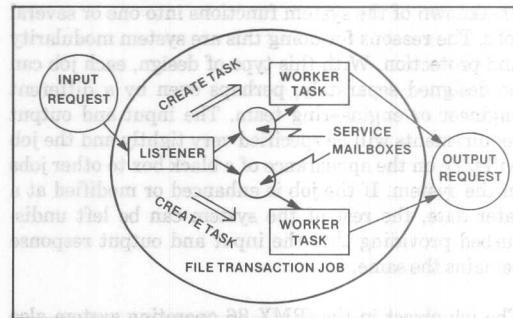


Figure 15. Diagram of Design of File Transaction Job

The string of transaction requests that follow will simply be demultiplexed by the listener task. The workstation ID will be searched for and, if found, the packet will be sent to the appropriate worker task. If a request comes in from a station that is not logged on, an error response is sent directly to the communications output mailbox for transmittal to the station that made the request.

If the request packet indicates that a station desires to log-off, the listener task will delete all local reference to the station and pass the packet along. The listener task cannot simply delete the worker since the worker may be in the process of servicing a previous I/O request. In general, it is never a good idea to arbitrarily delete another task. A better protocol is to pass along the message signaling the worker task to delete itself when convenient.

An investigation of the intertask communications needs highlights the requirement for passing data between tasks. The interjob communications protocol discussed earlier specified that the listener task will receive input request segments from the communications job via a mailbox.

Within these segments are fields containing the workstation ID and the command. Based upon these fields one of two things happens. If the command indicates that the station wishes to log on, a new worker task must be created to process the I/O requests that will follow.

The code executed by all worker tasks will be identical since they all perform identical functions. However, some unique pieces of information must be passed to a new worker task. This can be accomplished by having the worker task first wait at a "log on" mailbox. Here it will receive a segment from the listener task which contains the necessary information (see Figure 16).

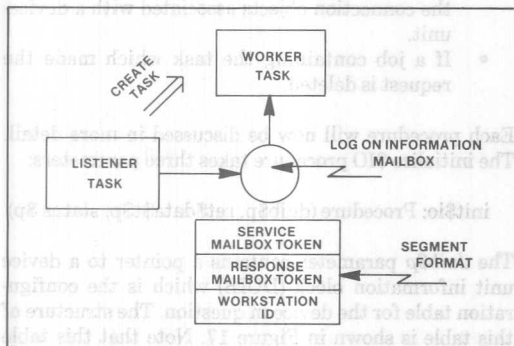


Figure 16. Communications Between Listener Task and a Newly Created Worker Task

After this initialization is complete, the workstation requests that are received by the listener task can be sent to the service mailbox associated with the workstation. The token for the service mailbox is one of the pieces of information contained in the log on segment.

The last communication path needed is predefined by the interjob communication protocol. When either the

listener task or one of the worker tasks needs to transmit a packet to a workstation, a segment is sent to the output request mailbox of the communication job.

The final step in the design methodology is to write program code for the tasks in the system. This step is performed in the Code Examples section.

APPLICATION EXAMPLE 2

This example will deal with the design of a custom device driver for the iRMX 86 operating system. As shown in Figure 6, a device driver accepts high-level commands from the file drivers (such as read, write, seek, etc.) and transforms these commands into I/O port read and write commands in order to communicate with the device itself. By studying the construction of a driver for the iSBC 534 Serial Communication Expansion Board, a better understanding of the iRMX 86 I/O system will be gained along with an example of the use of nucleus facilities to construct a higher-level software function.

Overview of Device Driver Construction

Each I/O device consists of a controller and one or more units. A device as a whole is identified by a device number. Units are identified by unit number and device-unit number. The unit number identifies the unit within the device and the device-unit number identifies the unit among all the units on all of the devices.

A device driver must be provided for every device in the hardware configuration. That device driver must handle the I/O requests for all of the units on the device. Different devices can use different device drivers; or if they are the same kind of device, they can use the same device driver code.

At its highest level, a device driver consists of four procedures which are called directly by the I/O System. These procedures can be identified according to purpose, as follows:

- Initialize I/O
- Finish I/O
- Queue I/O
- Cancel I/O

When a user makes an I/O System call to manipulate a device, the I/O System ultimately calls one or more of these procedures, which operate in conjunction with an interrupt handler to coordinate the actual I/O transfers. This section provides a general description of each of these procedures, and the interrupt handler.

INITIALIZE I/O

This procedure creates all of the iRMX 86 objects needed by the remainder of the routines in the device driver. It typically creates an interrupt task and a segment to store data local to the device. It also performs device initialization, if any such is necessary. The I/O System calls this routine just prior to the first attach of a unit on the device (the first RQ\$A\$PHYSICAL\$ATTACH\$DEVICE system call). The time sequence of calls to these procedures will be described a little later.

FINISH I/O

The I/O System calls this procedure after all units of the device have been detached (the last RQ\$A\$PHYSICAL\$DETACH\$DEVICE system call). The *finish*\$I/O procedure performs any necessary final processing on the device and deletes all of the objects used by the device handler, including the interrupt task and the device-local data segment.

QUEUE I/O

This procedure places I/O requests on a queue, so that they can start when the appropriate unit becomes available. If the device is not busy, the *queue*\$I/O procedure starts the request.

CANCEL I/O

This procedure cancels a previously queued I/O request. Unless the device is such that a request can take an indefinite amount of time to process (such as keyboard input from a terminal), this procedure can perform a null operation.

INTERRUPT HANDLERS AND INTERRUPT TASKS

After a device finishes processing an I/O request, it sends an interrupt to the iRMX 86 system. As a consequence, the interrupt handler for the device is called. This handler either processes the interrupt itself or signals an interrupt task to process the interrupt. Since an interrupt handler is limited in the types of system calls that it can make, an interrupt task usually services the interrupt. The interrupt task feeds the results of the interrupt back to the application software (data from a read operation, status from other types of operations). It then gets the next I/O request from the queue and starts the device processing this request. This cycle continues until the device is detached. The interrupt task is normally created by the *initialize* I/O procedure.

The I/O System calls each one of the four device driver procedures in response to specific conditions. Three of the procedures are called under the following conditions:

- 1) In order to start I/O processing, the user must make an I/O request. This can be done by making a variety of system calls. However, the first I/O request to each device-unit must be the RQ\$A\$PHYSICAL\$ATTACH\$DEVICE system call.
- 2) The I/O System checks to see if the I/O request results from the first RQ\$A\$PHYSICAL\$ATTACH\$DEVICE system call for the device (the first unit attached in a device). If it is, the I/O System realizes that the device has not been initialized and calls the *initialize* I/O procedure first, before queueing the request.
- 3) Whether or not the I/O System called the *initialize* I/O procedure, it calls the *queue* I/O procedure to queue the request for execution.
- 4) The I/O System checks to see if the request just queued resulted from the last RQ\$A\$PHYSICAL\$DETACH\$DEVICE system call for the device (detaching the last unit of a device). If so, the I/O System calls the *finish* I/O procedure to do any final processing on the device and clean up objects used by the device driver routines:

The I/O System calls the fourth device driver procedure, the *cancel* I/O procedure, under the following conditions:

- If the user makes an RQ\$A\$PHYSICAL\$DETACH\$DEVICE system call specifying the hard detach option, in order to forcibly detach the connection objects associated with a device-unit.
- If a job containing the task which made the request is deleted.

Each procedure will now be discussed in more detail. The *initialize* \$I/O procedure takes three parameters:

init\$io: Procedure (duib\$p, ret\$data\$t\$p, status \$p)

The *duib*\$p parameter contains a pointer to a device unit information block (DUIB) which is the configuration table for the device in question. The structure of this table is shown in Figure 17. Note that this table contains pointers to device and unit information tables which can contain hardware specific information (such as I/O base addresses, interrupt levels etc.).

The second parameter is a pointer to a word which can be assigned the value of a token for an iRMX 86 object. Quite often this object would be a segment which could be created by the *init*\$io procedure and filled with information needed by the other procedures in the driver. The token for this segment will be provided to the other procedures when they are called.

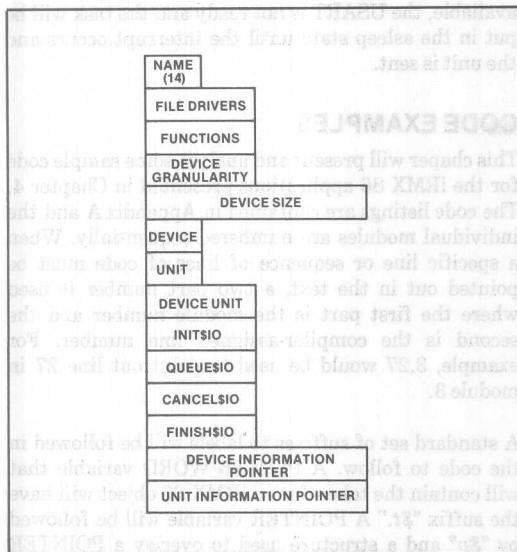


Figure 17. DUIB Format

The final argument in the call is a pointer to a status word. This word should be assigned by the *init\$io* procedure before a RETURN is executed. If a non-zero value is returned indicating an error condition, the I/O System assumes that *init\$io* has deleted any objects created before the error was encountered.

The *finish\$io* procedure is called by the I/O System just after the last *detach\$device* call is made on the device. This procedure is expected to delete any objects created by the *init\$io* procedure and shut down the connected device.

finish\$io: Procedure (duib\$p, ret\$data\$t);

Once again, the first parameter to the call is a pointer to a DUIB. The second parameter is the token returned by the *init\$io* procedure.

The *queue\$io* procedure is called to initiate an I/O request.

queue\$io: Procedure (IORS\$t, duib\$p, ret\$data\$t)

The specifics of the request are indicated in an I/O request segment (IORS) which is provided by the first parameter. The format of this segment is shown in Figure 18. The most important fields here are the count, function, status and buffer pointer fields which tell the *queue\$io* procedure what needs to be done. The second and third parameters are once again the pointer to the DUIB and the token for the object

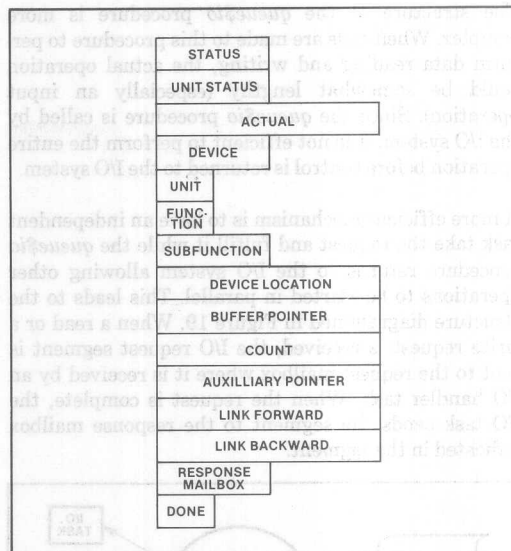


Figure 18. I/O Request Segment Format

created by the *init\$io* procedure.

The final device driver procedure is *cancel\$io*. This procedure is called by the I/O System to cancel a previous I/O request. If the device is of such a nature that a request will complete in a bounded amount of time, this procedure can be a null procedure. The parameters to the call are identical to those for the *queue\$io* call.

In addition to the elementary support discussed here, the I/O System provides extra support to the designer of a device driver if some simplifying assumptions about the device can be made. Also, if the device supports random access (such as disks, magnetic bubbles, etc.), support routines can be used to simplify the process of blocking and deblocking I/O requests. More detail on the process of writing I/O drivers can be found in the manual titled "A Guide to Writing Device Drivers for the iRMX 86 I/O System."

Design of an iSBC 534™ Device Driver

The following section will discuss an example device driver for the iRMX 86 Operating System. The driver will be for the iSBC 534 board which contains four 8251 USART devices; therefore, there is one device and four units on the device.

The *init\$io* procedure for this driver initializes the hardware, creates an interrupt task, creates other necessary objects and creates a segment to contain the relevant information.

The structure of the *queue\$io* procedure is more complex. When calls are made to this procedure to perform data reading and writing, the actual operation could be somewhat lengthy (especially an input operation). Since the *queue\$io* procedure is called by the I/O system, it is not efficient to perform the entire operation before control is returned to the I/O system.

A more efficient mechanism is to have an independent task take the request and fulfill it while the *queue\$io* procedure returns to the I/O system allowing other operations to be started in parallel. This leads to the structure diagrammed in Figure 19. When a read or a write request is received, the I/O request segment is sent to the request mailbox where it is received by an I/O handler task. When the request is complete, the I/O task sends the segment to the response mailbox indicated in the segment.

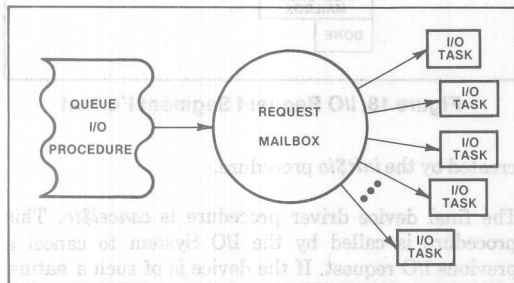


Figure 19. Queue\$io Procedure Interface to I/O Tasks

The remaining design of the device driver is concerned with interrupt handling. The iSBC 534 board contains four 8251 USART devices. Each device supplies two interrupts; one indicating that the receiver has a data character available and the other indicating that the transmitter is ready to accept a character. Each of these interrupts (8 in all) are connected to one of the 8259 Interrupt Controllers on the board. The software on the iSBC 86/12A board must read a register in the 8259 controller to determine which of the eight sources caused the current interrupt. This information must then be fed to the I/O task which may be waiting for the event.

One way to meet this requirement uses an interrupt task for the iSBC 534 board. The task receives the interrupt, determines which device caused it, and sends a unit to a semaphore to indicate the occurrence of the event. Thus, when an I/O task wishes to be informed of a receiver or transmitter interrupt, it simply tries to receive a unit from the appropriate semaphore. If a unit is available, the receiver has a character or the transmitter is ready. If the unit is not

available, the USART is not ready and the task will be put in the asleep state until the interrupt occurs and the unit is sent.

CODE EXAMPLES

This chapter will present and analyze some sample code for the iRMX 86 applications presented in Chapter 4. The code listings are contained in Appendix A and the individual modules are numbered sequentially. When a specific line or sequence of lines of code must be pointed out in the text, a two part number is used where the first part is the module number and the second is the compiler-assigned line number. For example, 3.27 would be used to point out line 27 in module 3.

A standard set of suffixes to labels will be followed in the code to follow. A PL/M-86 WORD variable that will contain the token for an iRMX 86 object will have the suffix "\$t." A POINTER variable will be followed by "\$p" and a structure used to overlay a POINTER allowing access to the base and offset will be followed by "\$p\$o."

Listener Task

The first module to be studied contains the code for the listener task. The various include statements bring in literal declarations and external procedure declarations. The file NUCPRM.EXT is on the iRMX 86 diskette and contains the external declarations for all iRMX 86 nucleus system calls.

Line 1.323 contains all of the declarations for the module. The literal *req\$segment\$struc* is used to access the fields of a segment returned from the communications job. The format of a request packet from a workstation is shown in Figure 20. The literal *node* is used to access the information in a segment used as a workstation descriptor in a list maintained by the listener task. The format of a *node* in this list is shown in Figure 21. The structure at the end of the declaration statement is used to individually access the two halves of a 32-bit PL/M-86 POINTER.

Note in line 1.330 that the task is coded as a public procedure having no parameters. A main procedure should never be used for a task's code since the preamble for a main procedure sets the stack pointer.

The mailbox to be used for sending a newly created worker task an information segment is called the *log\$on\$info\$mbx*. This mailbox is created in line 1.331. Lines 1.332-1.334 perform the operation of finding the tokens for the communication job's input and output request mailboxes in the object directory of

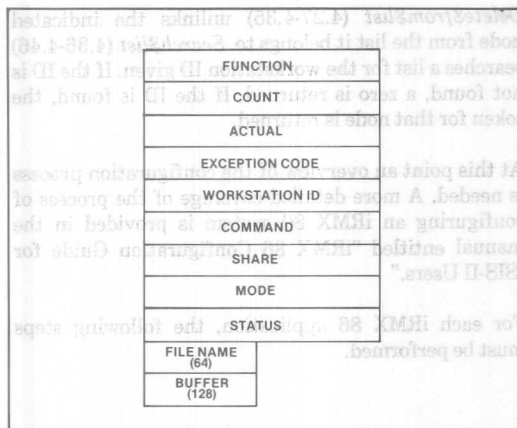


Figure 20. Request Packet Format

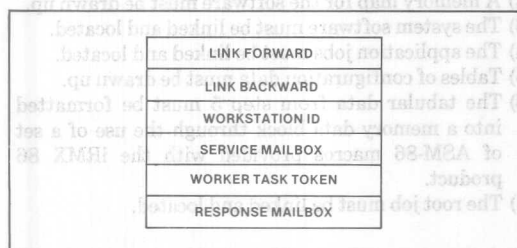


Figure 21. Workstation Descriptor Format

the root job. The token for the root job is obtained by the system call in 1.332.

Whenever a workstation logs on, various actions are taken by the listener task. One of these actions involves adding a descriptor for the workstation to a list so that the state of the workstation can be maintained by the listener task. The list structure is shown in Figure 22. Statements 1.336-1.340 create the root of this list and initialize the list to an empty state.

Line 1.340 marks the beginning of an infinite loop. Most often a task executes a procedure which performs some initialization and then enters an endless loop performing the necessary processing. The literal "forever" translates into "while 1."

A packet is received from the input mailbox by the call in line 1.341. The command field of the message is checked in line 1.343. If the command indicates that a log on request is being made, lines 1.345-1.356 are executed. A log on information segment is created in line 1.345. A mailbox is created to handle further request packets and another is created to be used by the worker task as a response mailbox. The worker

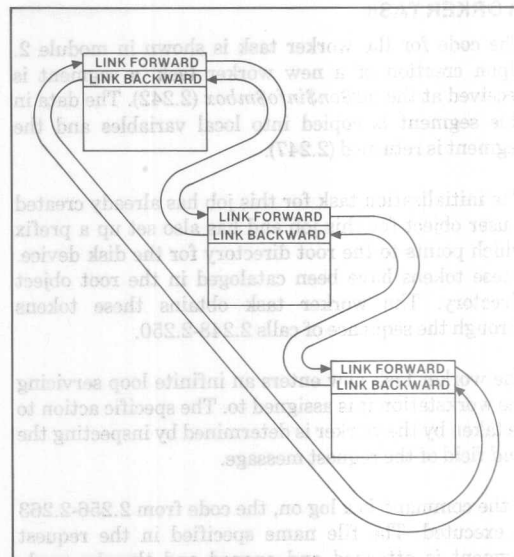


Figure 22. Workstation Descriptor List Structure

task that will handle I/O requests from this workstation is created in line 1.351. Note the use of the structure *data\$seg\$p\$o*, which is declared at the same address as the POINTER *data\$seg\$p*. The POINTER is initialized to equal the beginning of the data segment of the worker task module (1.323) and then the base portion is used as a parameter in the create task call.

Once the worker task is created, it will wait at the *log\$on\$info\$mbox* for a segment giving it its initialization information. The segment is sent in line 1.352 and received back as an acknowledgement in line 1.353. At this point, the segment is inserted on the list of active workstation descriptors by the call in line 1.354. Finally the request packet itself is sent to the worker task via the service mailbox for the new worker.

If a log off request is received, lines 1.358 to 1.366 are executed. First, the active workstation list is searched for the ID of the requesting station. If the station is not found to be logged on, the status field is set and the request segment is sent to the workstation through the communications job. If the station is logged on, the descriptor is deleted from the list, the packet is sent along to the worker task, and the descriptor is deleted.

If the command is anything but log on or log off, lines 1.368-1.376 are executed. Once again the station ID is checked to see if it is logged on. If not, an error message is returned. If the station is logged on, the request packet is sent along to the worker task.

WORKER TASK

The code for the worker task is shown in module 2. Upon creation of a new worker task, a segment is received at the *log\$on\$info\$mbx* (2.242). The data in this segment is copied into local variables and the segment is returned (2.247).

The initialization task for this job has already created a user object for this job and has also set up a prefix which points to the root directory for the disk device. These tokens have been cataloged in the root object directory. The worker task obtains these tokens through the sequence of calls 2.248-2.250.

The worker task now enters an infinite loop servicing the workstation it is assigned to. The specific action to be taken by the worker is determined by inspecting the *cmd* field of the request message.

If the command is a log on, the code from 2.256-2.263 is executed. The file name specified in the request segment is attached and opened and thereby made ready for subsequent I/O requests. After this, an acknowledgement is sent back to the workstation via the *output\$request\$mailbox* (2.263).

If a log off command is received, the file is closed and detached, the service and response mailboxes are deleted, a response is sent to the workstation and the worker task is deleted.

If the command is either a read or write command, the operation is performed by calling the I/O system. When the response is received, an acknowledgement is sent to the workstation. Note that the task would normally perform more processing. In this example its duties have been kept simple.

POINTERIZE PROCEDURE

The ASM-86 code for the *pointerize* support routine is shown in Module 3. The token for a segment is the base portion of a 32-bit POINTER to the memory. In order to access the data in a segment, this 16-bit token must be loaded into the base part of a POINTER while the offset portion of the POINTER is set to zero. The base and offset values are returned in the ES and BX registers as specified by the PL/M-86 calling conventions. This is the operation performed by the *pointerize* routine.

LIST MANIPULATION ROUTINES

Lines 4.1-4.47 provide three subroutines used by the tasks in this system to manipulate the list of workstation descriptors. *Insert\$on\$list* (4.15-4.26) inserts the indicated node at the head of the list whose root is given as the first parameter.

Delete\$from\$list (4.27-4.35) unlinks the indicated node from the list it belongs to. *Search\$list* (4.36-4.46) searches a list for the workstation ID given. If the ID is not found, a zero is returned. If the ID is found, the token for that node is returned.

At this point an overview of the configuration process is needed. A more detailed coverage of the process of configuring an iRMX 86 system is provided in the manual entitled "iRMX 86 Configuration Guide for ISIS-II Users."

For each iRMX 86 application, the following steps must be performed.

- 1) Program code for each task in the system must be written and compiled or assembled.
- 2) A memory map for the software must be drawn up.
- 3) The system software must be linked and located.
- 4) The application jobs must be linked and located.
- 5) Tables of configuration data must be drawn up.
- 6) The tabular data from step 5 must be formatted into a memory data block through the use of a set of ASM-86 macros provided with the iRMX 86 product.
- 7) The root job must be linked and located.

The code executed by the root task is part of the iRMX 86 system code. This task is initially the only task in the system. The root task will access the data block constructed by the ASM-86 macros and will create the user jobs specified by the macros. The data for the configuration process for example 1 is shown in Appendix B.

The first page diagrams the memory map for the example. The iterative link and locate process to put these pieces together begins on the second page. The LINK86 and LOC86 commands shown place the iRMX/86 nucleus into memory. The LOCATE map indicates that the last memory location used by the nucleus was 077DFH. Therefore, the next contiguous piece, the I/O system, is located at 077EOH.

This process is repeated for the remainder of the jobs in the system.

When the link and locate process is complete, the information for the ASM-86 macros must be brought together. Worksheets are provided in the iRMX 86 configuration guide to simplify this process.

The filled-out worksheets for the macros are shown in the appendix. A configuration file is constructed using

the editor and the worksheet information is entered into this file. When the file is complete, the configuration table is created by assembling the file CTABLE.A86. This file accesses the configuration file built earlier.

The configuration tables are then linked and located together with the code for the root task and the system generation process is complete.

EXAMPLE 2

INIT\$IIO AND FINISH\$IIO

The *start\$and\$finish* module (5.1-5.371) contains the code for the *init\$534\$Iio* and *finish\$534\$Iio* procedures. The *init\$534\$Iio* procedure creates a segment, shown in Figure 23, which is used to hold the various pieces of information needed by the other driver procedures (5.323). The discussion of this procedure in Chapter 4 pointed out that any errors encountered in the initialization are indicated by the non-zero status and that the assumption is made that any partial creations must be cleaned up by the *init\$Iio* procedure. This assumption is carried out by the check at line 5.324 (and the others at 5.331, 5.335, 5.339 and 5.342).

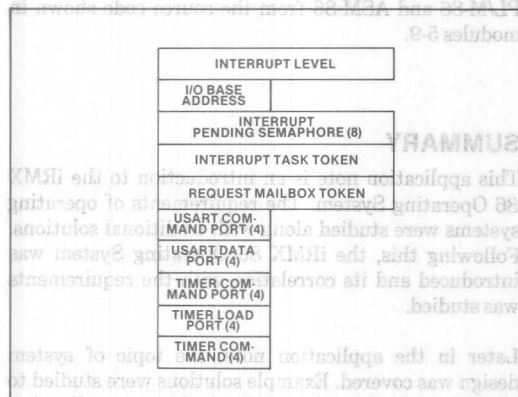


Figure 23. *init\$534\$Iio* Segment Format

The device information contained in the device unit information block for this device is retrieved in line 5.328-5.329. A mailbox to be used for sending I/O request segments to the I/O handler tasks is created in line 5.330. The interrupt task for this job is created by the call in line 5.337.

The *do loop* starting at line 5.340 is executed to create eight semaphores to be used by the interrupt task to indicate the occurrence of an interrupt. Note that the initial value of the semaphore is zero (no interrupt

pending) and the maximum value is one. Since the nature of the 8251 USART device does not support buffering, when a new character overruns the previous character before the interrupt can be serviced, the data is lost. Therefore, there is no need to indicate the occurrence of multiple interrupts pending on the same device.

The call at line 5.345 initializes the programmable devices on the iSBC 534 board. If execution has proceeded to line 5.346, the initialization is complete and a zero status is returned. If an error occurred at any point, the code in lines 5.348-5.356 will clean up the partial initialization.

The *finish\$534\$Iio* procedure (5.358-5.370) undoes the work of the *init\$534\$Iio* procedure. The segment, mailbox, interrupt task and semaphores are all deleted.

The *queue\$534\$Iio* procedure is shown in lines 6.1-6.382. In line 6.322 the function field of the I/O request segment is checked to see if it is within bounds. If it is not, a bad status code is returned. If the function is valid, a *do case* block is executed using the function code as the index.

If a read request is encountered, the auxiliary pointer is set to point to the *ret\$data* structure (initialized earlier by the *init\$534\$Iio* procedure). In line 6.327 the segment is then sent to the request mailbox to be received and processed by an I/O processor task. In lines 6.330-6.334 the same action is taken with write requests.

Since this driver does not support seeking and special functions, the code for these two cases simply returns an error condition.

In the case of an *attach\$device* call, the code in lines 6.341-6.361 is executed. First, two I/O processing tasks are created. All of these tasks execute identical code and each task is capable of servicing a read or a write request on any 8251. Two tasks are created for each 8251 device so that the peak load can always be handled (that is, all receivers and transmitters going simultaneously). Lines 6.346-6.357 perform the initialization of the 8251 USART and the baud rate generators for this channel. The calls in line 6.358 and 6.359 accept an interrupt and a character from the semaphore associated with the receiver just initialized. This is done to clear off an interrupt generated by the 8251 whenever it is initialized.

In the case of a *detach\$device* call, the code in lines 6.363-6.367 sends the I/O request segment to the

request mailbox twice. This is done to signal two of the I/O handler tasks to delete themselves. As discussed earlier in the *attach\$device* section, none of the I/O handler tasks is any different from any of the others. There are two created for each 8251 device which is attached. The protocol set up for their deletion is shown here. When an I/O handler task receives a segment of type "*detach\$device*" it will send the segment to the response mailbox and then delete itself.

The code for the open and close requests is the same. Both cases are supported but are NOPs since no specific action needs to be taken by the driver.

Lines 6.379-6.382 contain the code for the *cancel\$534\$io* procedure. As discussed earlier, this procedure is simply a placeholder and serves no particular purpose.

INTERRUPT CONTROL MODULE

The interrupt handler and interrupt task are shown in lines 7.1-7.329. The interrupt task is the first piece executed. It is created by the *init\$534\$io* procedure. It calls *RQ\$set\$interrupt* in line 7.325 to indicate to the iRMX 86 nucleus that it is an interrupt task.

Once the initialization is complete, the task enters an infinite loop. The call to *RQ\$wait\$interrupt* in line 7.322 causes the task to be put into the asleep state until an interrupt occurrence is signaled. The task will be returned to the READY state when an interrupt occurs, the interrupt handler is started, and the call to *RQ\$signal\$interrupt* is executed at line 7.312. The current interrupt level is then determined by polling the 8259 chip on the iSBC 534 board. Using the encoded level number, a unit is sent to the appropriate semaphore to indicate that an interrupt is pending.

I/O TASK

The final procedure that makes up this driver contains the code for the tasks that perform the actual I/O to the iSBC 534 board. The loop executed by each task starts by waiting at the request mailbox for an I/O request segment. When the segment is sent by the *queue\$534\$IO* procedure, its function code is checked (line 8.327, 8.332, 8.340). If the function is *f\$detach\$device*, the task sends the segment to the response mailbox and then deletes itself.

If the request was for a read, the task fills the buffer with input data. The call at line 8.334 waits for a unit at the semaphore which will indicate a receiver ready on the input line. When the unit is sent by the interrupt task, the character is read in, the pointers and counts are updated, and another unit is requested.

The last request which is recognized by the I/O task is for a write operation. The code for this request is almost identical to the code for a read request. An interrupt from the transmitter is awaited, a character is output and the counts are updated in lines 8.341-8.346.

Once the request is fulfilled, the message is sent to the response exchange in line 8.350.

The configuration of this system is studied next. The code for the iSBC 534 driver is linked directly to the rest of the I/O system libraries. The entry point addresses for the *queue\$534\$io*, *cancel\$534\$io*, *init\$534\$io*, and *finish\$534\$io* procedures are declared in the IOCNFG.A86 file on the I/O system disk. This file also contains the device unit information block (DUIB) structures for the four units on the iSBC 534 board. The unique information for the iSBC 534 device and the units on the device is contained in the device and unit information tables. Pointers to these tables are contained in the DUIB structures. All of this information is shown in Figure 24.

The submit file used to build an I/O system using the iSBC 534 driver is shown in Figure 25. The file DRV534.LIB contains the object files generated by PL/M-86 and ASM-86 from the source code shown in modules 5-9.

SUMMARY

This application note is an introduction to the iRMX 86 Operating System. The requirements of operating systems were studied along with traditional solutions. Following this, the iRMX 86 Operating System was introduced and its correlation with the requirements was studied.

Later in the application note, the topic of system design was covered. Example solutions were studied to solidify a methodology for solving application problems and then the code for these solutions was discussed to gain insight into the details of implementing iRMX 86 systems.

The purpose of a configurable, real-time, multi-purpose operating system is to provide a solid foundation for application software. The iRMX 86 system provides this foundation, giving the software engineer a means to quickly and easily implement new designs. In addition, the iRMX 86 architecture is the bridge to future technology providing the designer with an upgrade path to future hardware and software products.

```

extrn  init534io: near
extrn  queue534io: near
extrn  cancel534io: near
extrn  finish534io: near

;
; Duib(8): iSBC 534, unit 1
;
define_duib <
&          'i534.1',          ; name (14)

&          03H,                ; sup$opt
&          00033H,             ; file drivers
&          0,                  ; granularity
&          0,0,                ; device size

&          3,                  ; device
&          1,                  ; unit
&          6,                  ; device_unit
&          init534io,          ; init$io
&          finish534io,        ; finish$io
&          queue534io,         ; queue$io
&          cancel534io,        ; cancel$io
&          dev 534 info,        ; device info
&          unit_534_1_info      ; unit_info
&>

; 534 device info
;
dev_534_info  dw  48H          ; level
               db  61          ; priority
               db  040H        ; base address

; unit info: iSBC 534.0
;
unit_534_0_info  db  4EH        ; usart$cmd
                 dw  8          ; baud rate

; unit info: iSBC 534.1
;
unit_534_1_info  db  4EH        ; usart$cmd
                 dw  8          ; baud rate

; unit info: iSBC 534.2
;
unit_534_2_info  db  4EH        ; usart$cmd
                 dw  8          ; baud rate

; unit info: iSBC 534.3
;
unit_534_3_info  db  4EH        ; usart$cmd
                 dw  8          ; baud rate

```

Figure 24. IOCNG A86 File Entries for iSBC 534™ Driver

```

; ios(date,origin)
;   Sample I/O System .csd file to link and locate an I/O System.
;
; This file links an I/O System with the timer included.
;
; This .csd file assumes the I/O System configuration module is
; iocnfg.a86 (found on the release diskette).
;
; The origin parameter sets the low address of the I/O System;
; all the segments are contiguous in memory.
;
asm86 :fl:iocnfg.a86 date(%0) print(:f5:iocnfg.lst)
link86 &
:fl:ios.lib(iocinit), &
:fl:iocnfg.obj, &
:fl:ios.lib, &
:fl:drv534.lib, &
:f4:rpifc.lib &
to :fl:ios.lnk map print(:f1:ios.mp1)
loc86 :fl:ios.lnk to :fl:ios map sc(3) print(:f1:ios.mp2) &
oc(noli,nopl,nocm,nosb) &
order(classes(code,data,stack,memory)) &
addresses(classes(code(%1))) &
segsz(stack(0))

```

Figure 25. Submit File for Generating an I/O System with the iSBC 534™ Driver

APPENDIX A

Code Listings

Module 1

ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE LISTENERMODULE
 OBJECT MODULE PLACED IN :F1:listen.OBJ
 COMPILER INVOKED BY: plm86 :F1:listen.plm PRINT(:F1:LISTEN.LST)
 DEBUG COMPACT OPTIMIZE(3) ROM DATE(5/28/80)

```

1      listener$module:
      do;

/*****

LISTENER: TASK.

This task creates segments, sends them to the input service
job to be filled with input packet info. Upon response
the info is checked to see what action needs to be taken.
If a log$on request is sensed, a worker task, service
mailbox, and response mailbox are created and the packet is
sent along to the worker task. If a log$off is sensed all
local reference to the workstation is deleted and the packet
is sent along to tell the worker to delete himself. If an
I/O request is sensed the station ID is checked to make
sure it is logged on. If it is, the packet is sent along to
the worker. If it isn't an error packet is sent back to the
requesting workstation.

*****/

$include(:f2:common.lit)
= $SAVE NOLIST
= $include(:f1:node.lit)
= /* literal declaration of node descriptor for list utilities */
=
11 1  declare
=     node literally 'structure(
=         link$f word,
=         link$b word,
=         work$station$ID word,
=         service$mbbox$T word,
=         worker$task$T word,
=         resp$mbbox$T word)';
= $include(:f1:lstutl.ext)
= /* external declarations for list manipulation utilities */
=
= $save nolist
= $include(:f1:pointtr.ext)
= /* external declaration of pointerize procedure */
= $save nolist
= $include(:f1:rqpckt.lit)
= /* literal declaration for request packet structure */
=
24 1  declare req$segment$struc literally 'structure(
=     funct word,
=     count word,
=     actual word,
=     ex$val word,
=     work$station$ID word,
=     cmd word,
=     share word,
=     mode word,
=     status word,
=     file$name (64) byte,
=     buf (128) byte)';
= $include(:f2:nucprm.ext)
= $SAVE NOLIST

321 1  worker$task: procedure external;
322 2  end worker$task;

```

Module 1, continued

```

323 1 declare
begin$listener$task$data byte public,
begin$worker$task$data byte external,
log$on$info$mbox$token public,
ex$val word,
log$on$mbox$name (7) byte data(6, 'LOG$ON'),
packet$size literally '132',
f$read literally '5',
f$write literally '6',
log$on literally '0',
log$off literally '1',
not$logged$on literally '1',
(root$job$token, input$request$mbox$token),
(output$request$mbox$token, resp$mbox$token),
(work$station$list$root$token, req$segment$token),
(log$on$info$seg$token, dummy$token, ws$desc$token),
(req$segment$ptr, work$station$list$root$ptr) pointer,
(log$on$info$seg$ptr, data$seg$ptr, ws$desc$ptr) pointer,
(req$segment based req$segment$ptr) req$segment$struct,
(work$station$list$root based work$station$list$root$ptr) node,
(log$on$info$seg based log$on$info$seg$ptr) node,
data$seg$ptr so structure(offset word, base word) at(@data$seg$ptr),
(ws$desc based ws$desc$ptr) node;

324 1 return$error$to$WS: procedure;

325 2 req$segment.func=f$write;
326 2 req$segment.status=not$logged$on;
327 2 call req$send$message(output$request$mbox$token, req$segment$token, 0, @ex$val);
328 2 return;
329 2 end;

330 1 Listener: procedure public; /* task */

331 2 log$on$info$mbox$token=rq$create$mailbox(0, @ex$val);
332 2 root$job$token=rq$get$task$tokens(3, @ex$val);
333 2 input$request$mbox$token=rq$lookup$object(
/* job */ root$job$token,
/* name */ @(9, 'INPUT$REQ'),
/* time limit */ 0FFFFH,
/* status ptr */ @ex$val);

334 2 output$request$mbox$token=rq$lookup$object(
/* job */ root$job$token,
/* name */ @(10, 'OUTPUT$REQ'),
/* time limit */ 0FFFFH,
/* status ptr */ @ex$val);

335 2 resp$mbox$token=rq$create$mailbox(0, @ex$val);
336 2 work$station$list$root$token=rq$create$segment(16, @ex$val);
337 2 work$station$list$root$ptr=pointerize(work$station$list$root$token);
338 2 work$station$list$root$link$ff=0;
work$station$list$root$link$b=work$station$list$root$token;
339 2 work$station$list$root.workstation$ID=0;

340 2 do forever;
341 3 req$segment$token=rq$receive$message(
/* mbox token */ input$request$mbox$token,
/* time limit */ 0FFFFH,
/* response ptr */ @dummy$token,
/* status ptr */ @ex$val);
342 3 req$segment$ptr=pointerize(req$segment$token);
343 3 if req$segment.cmd= log$on then
344 3 do;

```

Module 1, continued

```

345 4      log$on$info$seg$task:=rq$create$segment(
          /* size */ 16,
          /* status ptr */ @ex$val);
346 4      log$on$info$seg$p=pointerize(
          log$on$info$seg$task);
347 4      log$on$info$seg.service$mbox$task:=
          rq$create$mailbox(0,@ex$val);
348 4      log$on$info$seg.resp$mbox$task:=
          rq$create$mailbox(0,@ex$val);
349 4      log$on$info$seg.work$station$ID=
          req$segment.work$station$ID;
350 4      data$seg$p=@begin$worker$task$data;
351 4      log$on$info$seg.worker$task$task:=
          rq$create$task(
          /* priority */ 200,
          /* start addr */ @worker$task,
          /* data seg ptr */ data$seg$p$base,
          /* stack pointer */ 0,
          /* stack size */ 500,
          /* task flags */ 0,
          /* status ptr */ @ex$val);
352 4      call rq$send$message(
          /* mbox token */ log$on$info$mbox$task,
          /* object token */ log$on$info$seg$task,
          /* response token */ resp$mbox$task,
          /* status ptr */ @ex$val);
353 4      log$on$info$seg$task:=rq$receive$message(
          /* mailbox token */ resp$mbox$task,
          /* time limit */ 0FFFFH,
          /* response token */ @dummy$task,
          /* status ptr */ @ex$val);
354 4      call insert$on$list(work$station$list$root$task,
          log$on$info$seg$task);
355 4      call rq$send$message(
          /* mbox tok */ log$on$info$seg.service$mbox$task,
          /* obj tok */ req$segment$task,
          /* response */ 0,
          /* status */ @ex$val);
356 4      end;
357 3      else if req$segment.cmd = log$off then
358 3      do;
359 4          ws$desc$task=search$list(work$station$list$root$task,
          req$segment.work$station$ID);
360 4          if ws$desc$task = 0 then
361 4              call return$error$to$WS;
          else
362 4              do;
363 5                  ws$desc$p=pointerize(ws$desc$task);
364 5                  call delete$from$list(
          ws$desc$task);
365 5                  call rq$send$message(
          ws$desc.service$mbox$task,
          req$segment$task,
          0,
          @ex$val);
366 5              end;
367 4          end;
          else
368 3          do;
369 4              ws$desc$task=search$list(work$station$list$root$task,
          req$segment.work$station$ID);

```

Module 1, continued

```

370 4      if ws$desc$=0 then
371 4      call return$error$to$WS;
      else
372 4      do;
373 5          ws$desc=pointerize(ws$desc$);
374 5          call rq$sendMessage(
              ws$desc.service$mbx$st,
              req$segment$,
              0,
              @ex$val);
      end;
375 5      end;
376 4      call rq$delete$segment(req$segment$,@ex$val);
377 3      end; /* of do forever */
378 3      end; /* of listener task */
379 2      end; /* of listener task */
380 1      end listener$module;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0281H      641D
CONSTANT AREA SIZE  = 0000H      0D
VARIABLE AREA SIZE  = 002BH      43D
MAXIMUM STACK SIZE  = 0018H      24D
694 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-86 COMPILATION.

Module 2

ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE WORKERTASK
 OBJECT MODULE PLACED IN :F1:worker.OBJ
 COMPILER INVOKED BY: plm86 :F1:worker.plm PRINT(:F1:WORKER.LST)
 DEBUG COMPACT OPTIMIZE(3) ROM DATE(5/28/80)

```

1      worker$task:
      do;

/*****

      WORKER$TASK: TASK.

      This module contains the code executed by the worker tasks.
      When started, the task goes to a mailbox to receive a segment
      containing initialization information. Using this information
      the task services a service mailbox performing any I/O functions
      requested of it. When a log$off request comes in the worker
      task closes and detaches the file and deletes itself.

      *****/

      $include(:f1:nucprm.ext)
=      $SAVE NOLIST
      $include(:f1:iosys.ext)
=      $save nolist
      $include(:f1:node.lit)
=      /* literal declaration of node descriptor for list utilities */
=      $save nolist
      $include(:f2:common.lit)
=      $SAVE NOLIST
      $include(:f1:pointtr.ext)
=      /* external declaration of pointerize procedure */
=      $save nolist
      $include(:f1:rqpckt.lit)
=      /* literal declaration for request packet structure */
=
=      $save nolist

239 1      declare
          read literally '1',
          write literally '5',
          log$on literally '2',
          log$off literally '3',
          (log$on$info$mbbox$token,output$request$mbbox$token) token external;

240 1      worker$task: procedure reentrant public;

241 2      declare
          (log$on$info$seg$token,log$on$resp$mbbox$token,resp$mbbox$token,
          root$job$token,user$object$token,prefix$token,iors$token,
          service$mbbox$token,conn$token,req$seg$token) token,
          (log$on$info$ptr,req$seg$ptr) pointer,
          (req$seg based req$seg$ptr) req$segment$struct,
          (log$on$info based log$on$info$ptr) node,
          (dummy$token,ex$val,work$station$ID) word;

242 2      log$on$info$seg$token=rq$receive$message(
          /* mbox token */      log$on$info$mbbox$token,
          /* time limit */      0FFFFH,
          /* response ptr */    @log$on$resp$mbbox$token,
          /* status ptr */      @ex$val);

243 2      log$on$info$ptr=pointerize(log$on$info$seg$token);
244 2      service$mbbox$token=log$on$info.service$mbbox$token;
245 2      resp$mbbox$token=log$on$info.resp$mbbox$token;
246 2      work$station$ID=log$on$info.work$station$ID;

```


Module 2, continued

```

247 2      call rq$sendMessage(
/* mbox token */ log$on$resp$mbox$t,
/* object token */ log$on$info$seg$t,
/* response token */ 0,
/* status ptr */ @ex$val);

248 2      root$job$t=rq$get$task$tokens(3,@ex$val);
249 2      user$object$t=rq$lookup$object(
/* job token */ root$job$t,
/* name */ @ (11,'USER$OBJECT'),
/* time limit */ 0FFFFH,
/* status ptr */ @ex$val);

250 2      prefix$t=rq$lookup$object(
/* job token */ root$job$t,
/* name */ @ (6,'PREFIX'),
/* time limit */ 0FFFFH,
/* status ptr */ @ex$val);

251 2      do forever;

252 3      req$seg$t=rq$receive$message(
/* mailbox token */ service$mbox$t,
/* time limit */ 0FFFFH,
/* response ptr */ @dummy$t,
/* status ptr */ @ex$val);

253 3      req$seg$p=pointerize(req$seg$t);

254 3      if req$seg.cmd=log$on then
255 3          do;
256 4          call rq$a$attach$file(
/* user object */ user$object$t,
/* prefix token */ prefix$t,
/* pathname */ @req$seg.file$name,
/* resp token */ resp$mbox$t,
/* status ptr */ @ex$val);
257 4          iors$t=rq$receive$message(
/* mbox token */ resp$mbox$t,
/* time limit */ 0FFFFH,
/* resp ptr */ @dummy$t,
/* status ptr */ @ex$val);

258 4          call rq$delete$segment(iors$t,@ex$val);
259 4          call rq$a$open(
/* connection */ conn$t,
/* mode */ req$seg.mode,
/* share */ req$seg.share,
/* resp token */ resp$mbox$t,
/* status ptr */ @ex$val);
260 4          iors$t=rq$receive$message(
/* mbox token */ resp$mbox$t,
/* time limit */ 0FFFFH,
/* resp ptr */ @dummy$t,
/* status ptr */ @ex$val);

261 4          call rq$delete$segment(iors$t,@ex$val);
262 4          req$seg.status=0;
263 4          call rq$sendMessage(
/* mbox token */ output$request$mbox$t,
/* object token */ req$seg$t,
/* resp ptr */ 0,
/* status ptr */ @ex$val);

264 4          end;

265 3      else if req$seg.cmd=log$off then
266 3          do;
267 4          call rq$a$close(
/* connection */ conn$t,
/* resp token */ resp$mbox$t,
/* status ptr */ @ex$val);

```

Module 2, continued

```

268 4      iorsSt= rq$receive$message(
/* mbox token */ resp$mboxSt,
/* time limit */ 0FFFFH,
/* resp ptr */ @dummySt,
/* status ptr */ @ex$val);
269 4      call rq$delete$segment(iorsSt,@ex$val);
270 4      call rq$a$delete$connection(
/* connection */ connSt,
/* response ptr */ resp$mboxSt,
/* status ptr */ @ex$val);
271 4      iorsSt=rq$receive$message(
/* mbox token */ resp$mboxSt,
/* time limit */ 0FFFFH,
/* response ptr */ @dummySt,
/* status ptr */ @ex$val);
272 4      call rq$delete$segment(iorsSt,@ex$val);
273 4      call rq$delete$mailbox(service$mboxSt,@ex$val);
274 4      call rq$delete$mailbox(resp$mboxSt,@ex$val);
275 4      req$seg.status=0;
276 4      call rq$send$message(
/* mbox token */ output$request$mboxSt,
/* object token */ req$segSt,
/* resp token */ 0,
/* status ptr */ @ex$val);
277 4      call rq$delete$task(0,@ex$val);
278 4      end;
279 3      else if req$seg.cmd=read then
280 3          do;
281 4              call rq$a$read(
/* connection */ connSt,
/* buf ptr */ @req$seg.buf,
/* count */ req$seg.count,
/* resp token */ resp$mboxSt,
/* status ptr */ @ex$val);
282 4              iorsSt=rq$receive$message(
/* mbox token */ resp$mboxSt,
/* time limit */ 0FFFFH,
/* resp ptr */ @dummySt,
/* status ptr */ @ex$val);
283 4              call rq$delete$segment(iorsSt,@ex$val);
284 4              req$seg.status=0;
285 4              call rq$send$message(
/* mbox token */ output$request$mboxSt,
/* object token */ req$segSt,
/* resp token */ 0,
/* status ptr */ @ex$val);
286 4              end;
287 3      else if req$seg.cmd=write then
288 3          do;
289 4              call rq$a$write(
/* connection */ connSt,
/* buf ptr */ @req$seg.buf,
/* count */ req$seg.count,
/* resp token */ resp$mboxSt,
/* status ptr */ @ex$val);
290 4              iorsSt=rq$receive$message(
/* mbox token */ resp$mboxSt,
/* time limit */ 0FFFFH,
/* resp ptr */ @dummySt,
/* status ptr */ @ex$val);
291 4              call rq$delete$segment(iorsSt,@ex$val);

```

Module 2, continued

```

292 4      call rq$send$message(
          /* mbox token */ output$request$mboxSt,
          /* object token */ req$segSt,
          /* resp token */ 0,
          /* status ptr */ @ex$val);
293 4      end;
          end; /* of do forever */
295 2      end; /* of task */
296 1      end worker$task;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0288H      648D
CONSTANT AREA SIZE  = 0000H      0D
VARIABLE AREA SIZE  = 0000H      0D
MAXIMUM STACK SIZE  = 0034H      52D
717 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-86 COMPILATION

Module 3

ISIS-II MCS-86 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE POINTR
 OBJECT MODULE PLACED IN: F1:POINTR.OBJ
 ASSEMBLER INVOKED BY: asm86 :f1:pointr.a86 debug pr(:f5:pointr.lst)

LOC	OBJ	LINE	SOURCE
		1	·\$title(pointerize Utility)
0004		2	arg_off equ 4 ; set args for "DELUXE"
----		3	
----		4	code segment word public 'CODE'
----		5	code ends
		6	
----		7	cgroup group code
----		8	code segment
		9	assume cs:cgroup
0000		10	
		11	pointerize proc near
		12	public pointerize
0000 55		13	push bp ; save
0001 8BEC		14	mov bp, sp ; mark stack
		15	
0004 []		16	token equ word ptr [bp + arg_off + 0]
		17	
0003 8E4604		18	mov es, token ; get base
0006 33DB		19	xor bx, bx ; zap offset
		20	
		21	; mov sp, bp ; restore stack
0008 5D		22	pop bp
0009 C20200		23	ret 2
		24	pointerize endp
----		25	code ends
		26	end

ASSEMBLY COMPLETE, NO ERRORS FOUND

Module 4

ISIS-II PL/M-86 X167 COMPILATION OF MODULE LISTUTILITIESMODULE
 OBJECT MODULE PLACED IN :F1:lstutl.OBJ
 COMPILER INVOKED BY: plm86 :F1:lstutl.plm PRINT(:F5:LSTUTL.LST)
 DEBUG COMPACT OPTIMIZE(3) ROM DATE(3/7/80)

```

1      list$utilities$module:
      do;

/*****
LIST$UTILITIES: PUBLIC PROCEDURES.

This module contains three list manipulation utilities.
Insert$on$list takes the given node and inserts it on the
list indicated by the root node parameter. Delete$from
list unlinks the indicated node from the list, it is
linked to. Search$list scans the list from the root looking
for the indicated node. If found, the token for the node
is returned. If not found, a zero is returned.

*****/

$include(:f4:common.lit)
= $SAVE. NOLIST
$include(:f1:node.lit)
= /* literal declaration of node descriptor for list utilities */
= $save nolist
$include(:f1:point.r.ext)
= /* external declaration of pointerize procedure */
= $save nolist

15 1      Insert$on$list: procedure( root$token,new$desc$token) reentrant public;
16 2          declare
              (root$token,new$desc$token,fwd$desc$token) token,
              (root$point,new$desc$point,fwd$desc$point) pointer,
              (root based root$point) node,
              (new$desc based new$desc$point) node,
              (fwd$desc based fwd$desc$point) node;

17 2          root$point=pointerize(root$token);
18 2          new$desc$point=pointerize(new$desc$token);
19 2          fwd$desc$token=root.link$token;
20 2          fwd$desc$point=pointerize(fwd$desc$token);
21 2          root.link$token=new$desc$token;
22 2          new$desc.link$token=fwd$desc$token;
23 2          new$desc.link$point=root$token;
24 2          fwd$desc.link$point=new$desc$token;
25 2          return;

26 2      end; /* insert$on$list */

27 1      Delete$from$list: procedure(desc$token) reentrant public;
28 2          declare
              desc$token token,
              (desc$point,b$desc$point,f$desc$point) pointer,
              (desc based desc$point) node,
              (b$desc based b$desc$point) node,
              (f$desc based f$desc$point) node;

29 2          desc$point=pointerize(desc$token);
30 2          b$desc$point=pointerize(desc.link$point);
31 2          f$desc$point=pointerize(desc.link$token);
32 2          b$desc.link$token=desc.link$token;
33 2          f$desc.link$point=desc.link$point;
34 2          return;

```


Module 4, continued

```

35 2      end; /* delete$from$list */
36 1      search$list: procedure(root$t, WSSID, word reentrant public;
37 2          declare
              (root$t, WSSID) word,
              (s$desc$p, root$p) pointer,
              (root based root$p) node,
              (s$desc based s$desc$p) node,
              s$desc$p$o structure (offset word, base word) at(@s$desc$p),
              temp pointer;

38 2          s$desc$p=pointerize(root$t);
39 2          next$node:
              if s$desc.work$station$ID=WSSID then
                  return s$desc$p$o.base;
40 2              if s$desc.link$f = root$t then
41 2                  return 0;
42 2              temp=pointerize(s$desc.link$f);
43 2              s$desc$p=temp;
44 2              goto next$node;
45 2          end; /* search$list */

46 2      end list$utilities$module;
47 1

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 00FEH      254D
CONSTANT AREA SIZE  = 0000H      0D
VARIABLE AREA SIZE  = 0000H      0D
MAXIMUM STACK SIZE  = 0018H      24D
114 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-86 COMPILATION

Module 5 SubN

ISIS-II PL/M-86 X167 COMPILATION OF MODULE STARTANDFINISH
 OBJECT MODULE PLACED IN :F1:STRFIN.OBJ
 COMPILER INVOKED BY: plm86::F1:STRFIN.plm PRINT(:F5:STRFIN.LST)
 DEBUG COMPACT OPTIMIZE(2) ROM DATE(4/28/80)

```

1      start$and$finish:
      do;
/*****
      INIT$534$I/O and FINISH$534$I/O: PUBLIC PROCEDURES.

      This module contains the init$534$I/O and the FINISH$534$I/O
      procedures which can be called by the RMX/86 I/O system. START$I/O
      is called just before the first attach$device is performed.
      It will create the interrupt task and the eight interrupt$pending
      semaphores. The FINISH$I/O procedure is called just after the
      last detach$device is performed. It undoes everything the START$I/O
      call did.

      *****/
      $include(:f4:nucprm.ext)
      = $SAVE NOLIST
      = $include(:f4:common.lit)
      = $SAVE NOLIST
      = $include(:f1:duib.lit)
      = /* duib structure definition */
      = $save nolist
      = $include(:f4:nerror.lit)
      =
      = $SAVE NOLIST
      = $include(:f1:pointtr.ext)
      = /* external declaration of pointerize procedure */
      = $save nolist
      = $include(:f1:ret$dt$lit)
      = /* literal declaration of ret$dt$ structure for init$534$I/O */
      = $save nolist

314 1      init$534$hw: procedure(data$P, external;
315 2          declare data$P pointer;
316 2      end init$534$hw; /* initializes 534 hardware */

317 1      int$534$task: procedure external;
318 2      end int$534$task;

319 1      declare
          begin$int$534$dt$ byte external,
          IO$base$addr byte public,
          int$level word public,
          g$ret$dt$P pointer public,
          req$mbx$st token public;

320 1      init$534$I/O: procedure(duib$P, ret$dt$st$P, status$P) reentrant public;

321 2      declare
          (duib$P, ret$dt$st$P, status$P) pointer,
          (duib$ based duib$P) dev$unit$info$block,
          (ret$dt$st$ based ret$dt$st$P) token,
          (status$ based status$P) word,
          dev$info$P pointer,
          dev$info$ based dev$info$P structure(
              level word,
              priority byte,
              IO$base$addr byte),
          call $delete$segment(ret$dt$st$P, I/O

```

```

data$seg$pointer,
data$seg$pointer structure(offset word,base word) at(@data$seg$pointer),
(i,j) byte;

322 2 declare
    ret$data$pointer,
    ret$data based ret$data$pointer structure(ret$data$pointer);

323 2 ret$data$pointer=rg$create$segment(size(ret$data),@ex$val);
324 2 if ex$val <> 0 then
325 2     goto err0;
326 2 g$ret$data$pointer,ret$data$pointer=pointerize(ret$data$pointer);
327 2 dev$pointer=duib.dev$pointer;
328 2 IO$base$addr,ret$data.IO$base=dev$pointer.IO$base$addr;
329 2 int$level,ret$data.int$level=dev$pointer.level;

/* create the request mailbox */
330 2 ret$data.request$mbx$pointer,req$mbx$pointer
    =rg$create$mbx(0,@ex$val);
331 2 if ex$val <> 0 then
332 2     goto err1;

333 2 ret$data.resp$mbx$pointer=rg$create$mbx(0,@ex$val);
334 2 if ex$val <> 0 then
335 2     goto err2; /* clean up partial creation */

336 2 data$seg$pointer=@begin$int$534$data;
337 2 ret$data.int$task$pointer=rg$create$task(
    /* priority */ dev$pointer.priority,
    /* entry point */ @int$534$task,
    /* data segment */ data$seg$pointer.base,
    /* stack pointer */ 0,
    /* stack size */ 400,
    /* task flags */ 0,
    /* status pointer */ @ex$val);
338 2 if ex$val <> 0 then
339 2     goto err3; /* can't create. clean up partial creation */

340 2 do i=0 to 7; /* create semaphores */
341 3     ret$data.int$sema(i)=rg$create$semaphore(
        /* initial value */ 0,
        /* max value */ 1,
        /* priority queue */ 1,
        /* status ptr */ @ex$val);
342 3 if ex$val <> 0 then
343 3     goto err4; /* clean up partial creation */

344 3 end;
345 2 call init$534$shw(ret$data$pointer);
346 2 status=ESOK;
347 2 return;

348 2 err4:
    do j=0 to i;
349 3     call rg$delete$semaphore(ret$data.int$sema(j),status$pointer);
350 3 end;
351 2 call rg$reset$interrupt(dev$pointer.level,status$pointer);
352 2 err3:
    call rg$delete$mbx(ret$data.resp$mbx$pointer,status$pointer);
353 2 err2:
    call rg$delete$mbx(ret$data.request$mbx$pointer,status$pointer);
354 2 err1:
    call rg$delete$segment(ret$data$pointer,status$pointer);

```

Module 5, continued

```

355 2      err0:
356 2          status=ex$val; /* restore original status condition */
357 2      return;
357 2      end; /* of procedure */

358 1      finish$534$I0: procedure(duib$P,ret$data$T) reentrant public;
359 2      declare
          duib$P pointer,
          dev$info$P pointer,
          dev$info$based*dev$info$P*structure(
              level word,
              priority byte,
              IO$base$addr byte),
          ret$data$P pointer,
          ret$data$based ret$data$P structure(ret$data$struct),
          (duib$based duib$P)dev$unit$info$block,
          ret$data$T token,
          i byte,
          ex$val word;

360 2      dev$info$P=duib.dev$info$P;
361 2      ret$data$P=pointerize(ret$data$T);
362 2      call rq$reset$interrupt(dev$info.level,@ex$val);
363 2      call rq$delete$mailbox(ret$data.request$mbx$T,@ex$val);
364 2      call rq$delete$mailbox(ret$data.resp$mbx$T,@ex$val);
365 2      do i=0 to 7;
366 3          call rq$delete$semaphore(
              ret$data.int$sema(i),
              @ex$val);
367 3      end; /* of procedure */
368 2      call rq$delete$segment(ret$data$T,@ex$val);
369 2      return;
370 2      end; /* of procedure */
371 1      end start$and$finish;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0220H      544D
CONSTANT AREA SIZE  = 0000H       0D
VARIABLE AREA SIZE  = 0009H       9D
MAXIMUM STACK SIZE  = 0034H      52D
671 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-86 COMPILATION

Module 6

ISIS-II PL/M-86 X167 COMPILATION OF MODULE QUEUE534IOMODULE
 OBJECT MODULE PLACED IN :F1:queio.OBJ
 COMPILER INVOKED BY: plm86 :F1:queio.plm PRINT(:F5:QUEIO.LST)
 DEBUG COMPACT OPTIMIZE(2) ROM DATE(4/25/80)

```

1      queue$534$io$module:
      do;

/*****
      QUEUE$534$IO. PUBLIC PROCEDURE.

      This procedure is called by the I/O System to queue
      an I/O request to the 534 board. The function field
      in the IORS is used to determine what specific action
      to take. Module also contains a dummy cancel$534$io
      procedure.
      *****/

      $include(:f4:nucprm.ext)
      $SAVE NOLIST
      $include(:f4:common.lit)
      $SAVE NOLIST
      $include(:f4:nerror.lit)

      $SAVE NOLIST
      $include(:f1:pointr.ext)
      /* external declaration of pointerize procedure */
      $save nolist
      $include(:f1:duib.lit)
      /* duib structure definition */
      $save nolist
      $include(:f1:iors.lit)
      /* literal declaration for iors */
      $save nolist
      $include(:f1:ret$dt$lit)
      /* literal declaration of ret$dt$ structure for init$534$io */
      $save nolist

315 1      io$534$task: procedure external;
316 2      end io$534$task;

317 1      declare
      begin$io$task$dt$ byte external;

318 1      queue$534$io: procedure(iors$dt,duib$sp,ret$dt$) reentrant public;

319 2      declare
      (iors$dt,ret$dt$) token,
      data$seg$sp pointer,
      data$seg$sp$so structure(offset word,base word) at(@data$seg$sp),
      IDDR literally '2AH',
      (duib$sp,ret$dt$,iors$sp) pointer,
      (duib based duib$sp) dev$unit$info$block,
      (ret$dt$ based ret$dt$) structure(ret$dt$),
      (iors based iors$sp) IO$request$result$segment,
      io$task$dt token,
      unit$info$sp pointer,
      unit$info based unit$info$sp structure(
          usart$cmd byte,
          baud$rate word),
      i byte,
      dummy$dt token,
      ex$dt$ word;
  
```


Module 6, continued

```

320 2      iors$sp=pointerize(iors$st);
321 2      ret$data$sp=pointerize(ret$data$st);
322 2      if iors.funct > 7 then
323 2          goto bad$request;
324 2      do case iors.funct;
325 3          do; /* case 0-- read */
326 4              iors.aux$sp=ret$data$sp;
327 4              call rq$send$message(
                  /* mbox */ ret$data.request$mbox$st,
                  /* token */ iors$st,
                  /* resp */ 0,
                  /* status ptr */ @ex$val);
328 4              return;
329 4          end;
330 3          do; /* case 1-- write */
331 4              iors.aux$sp=ret$data$sp;
332 4              call rq$send$message(
                  /* mbox */ ret$data.request$mbox$st,
                  /* token */ iors$st,
                  /* resp */ 0,
                  /* status ptr */ @ex$val);
333 4              return;
334 4          end;
335 3          do; /* case 2--seek (illegal) */
336 4              goto bad$request;
337 4          end;
338 3          do; /* case 3-- special (illegal) */
339 4              goto bad$request;
340 4          end;
341 3          do; /* case 4-- attach$device */
              /* create two I/O tasks */
342 4              data$seg$sp=@begin$IIO$task$data;
343 4              do i=0 to 1;
344 5                  io$task$st= rq$create$task(
                      /* priority */ 150,
                      /* entry pnt */ @io$534$task,
                      /* data seg */ data$seg$sp$.base,
                      /* stack ptr */ 0,
                      /* stack size */ 500,
                      /* task flags */ 0,
                      /* status ptr */ @ex$val);
345 5              end;
346 4              unit$info$sp=duib.unit$info$sp;
347 4              do i=0 to 3;
348 5                  output(ret$data.usart$cmd$port(iors.unit))=0;
349 5              end;
350 4              output(ret$data.usart$cmd$port(iors.unit))=40H;
351 4              output(ret$data.usart$cmd$port(iors.unit))=
                  unit$info.usart$cmd;
352 4              output(ret$data.usart$cmd$port(iors.unit))=27H;
353 4              output(ret$data.IO$base+0CH)=0; /* select ctrl blk */
354 4              output(ret$data.timer$cmd$port(iors.unit))=
                  ret$data.timer$cmd(iors.unit);
355 4              output(ret$data.timer$load$port(iors.unit))=
                  low(unit$info.baud$rate);
356 4              output(ret$data.timer$load$port(iors.unit))=
                  high(unit$info.baud$rate);

```

Module 6, continued

```

357 4      output(ret$data.IO$base+0DH)=0; /* select data blk */
/* accept interrupt and character from receiver */

358 4      dummySt=rq$receive$units(
/* sema */ ret$data.int$sema( 2 * iors.unit),
/* units */ 1,
/* time$out */ 0,
/* status */ @ex$val);
359 4      i=input(ret$data.usart$data$port(iors.unit));
360 4      goto ok$send$resp;
361 4      end;

362 3      do; /* case 5-- detach$device */

/* send two copies of the detach request to the request mailbox.
This will signal to two of the I/O tasks that they are to
delete themselves */

363 4      call rq$send$message(
/* mbox token */ ret$data.request$mboxSt,
/* object token */ iorsSt,
/* response */ ret$data.resp$mboxSt,
/* status */ @ex$val);
364 4      dummySt=rq$receive$message(
/* mbox token */ ret$data.resp$mboxSt,
/* time$limit */ 0FFFFH,
/* response ptr */ @dummySt,
/* status ptr */ @ex$val);
365 4      call rq$send$message(
/* mbox token */ ret$data.request$mboxSt,
/* object token */ iorsSt,
/* response */ ret$data.resp$mboxSt,
/* status */ @ex$val);
366 4      dummySt=rq$receive$message(
/* mbox token */ ret$data.resp$mboxSt,
/* time$limit */ 0FFFFH,
/* response ptr */ @dummySt,
/* status ptr */ @ex$val);
367 4      goto ok$send$resp;
368 4      end;

369 3      do; /* case 6-- open */
370 4      goto ok$send$resp;
371 4      end;

372 3      do; /* case 7-- close */
373 4      goto ok$send$resp;
374 4      end;
375 3      end; /* do case */
376 2      return;
377 2      bad$request:
iors.status=IDDR;
378 2      goto send$resp;
379 2      ok$send$resp:
iors.status=ESOK;
380 2      send$resp:
call rq$send$message(iors.resp$mbox,iorsSt,0,@ex$val);
381 2      return;
382 2      end; /* procedure */

383 1      cancel$534$io: procedure(iorsSt,duib$p,ret$dataSt) public;
384 2      declare
(iorsSt,ret$dataSt) token,
duib$p pointer;

385 2      return;

```

Module 6, continued

```

386 2      end;
387 1      end queue$534$io$module;
MODULE INFORMATION:

```

```

CODE AREA SIZE      = 020CH      524D
CONSTANT AREA SIZE  = 0000H      0D
VARIABLE AREA SIZE  = 0000H      0D
MAXIMUM STACK SIZE  = 0038H      56D
729 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-86 COMPILATION

This module contains the interrupt handler and the interrupt task for the 534 board interrupt. The handler simply calls signal\$interrupt and the task reads the ISR on the 534 board's 8328 and sends a unit to one of eight interrupt pending semaphores to signal the occurrence of the event.

```

*****
include("l3:inuprm.ext")
$NOI$ NOI$
include("l3:retbcs.lit")
/* literal declaration of retbcs structure for int$534 */
$NOI$ NOI$
include("l3:common.lit")
$NOI$ NOI$

308 1 declare
begin int$534$byte public,
q$ret$534$pointer external,
IO$pass$534$byte external,
int$534$word external;

309 1 int$534$task: procedure interrupt 5;

310 2 declare
1 word,
ex$val word;

311 3 1=rd$534$level(ex$val);
312 3 call rd$534$interrupt(1,ex$val);
313 3 return;
314 3 end;

315 1 int$534$task: procedure reentr$ public;

316 2 declare
IO$534$base byte,
int$534$level word,
ret$534$pointer,
ret$534$base: structure(retr$534$ptr),
c$level byte,
ex$val word,
sol literally '30H';

317 3 IO$534$base=IO$pass$534$ptr;
318 3 int$534$level=int$534$level;
319 3 ret$534$base=ret$534$ptr;
320 3 call rd$534$interrupt(
/* level */ int$534$level,
/* flags */ 1,
/* entry point */ int$534$ptr(int$534$base),
/* data segment */ 0,
/* status ptr */ ex$val);

```

OBJECT MODULE PLACED IN :F1:int534.OBJ
 COMPILER INVOKED BY: plm86 :F1:int534.plm PRINT(:F1:INT534.LST)
 DEBUG COMPACT OPTIMIZE(2) ROM DATE(5/28/80)

```

1      $nointvector
      Interrupt$534$module:
      do;

/*****

      INT$534$TASK and INT$534$HND:
      PUBLIC PROCEDURES:

      This module contains the interrupt handler and the interrupt
      task for the 534 board interrupt. The handler simply calls
      signal$interrupt and the task reads the ISR on the 534
      board's 8259 and sends a unit to one of eight interrupt$
      pending semaphores to signal the occurrence of the event.

*****/

      $include(:f2:nucprm.ext)
=      $$SAVE NOLIST
      $include(:f1:retddta.lit)
=      /* literal declaration of ret$data structure for init$534$io */
=      $$save nolist
      $include(:f2:common.lit)
=      $$SAVE NOLIST

308 1      declare
          begin$int$534$data byte public,
          g$ret$data$p pointer external,
          IO$base$addr byte external,
          int$level word external;

309 1      int$534$hnd: procedure interrupt 5;

310 2      declare
          1 word,
          ex$val word;

311 2      l=rq$get$level(@ex$val);
312 2      call rq$signal$interrupt(1,@ex$val);
313 2      return;
314 2      end;

315 1      int$534$task: procedure reentrant public;

316 2      declare
          IO$534$base byte,
          int$534$level word,
          ret$data$p pointer,
          ret$data based ret$data$p structure(ret$data$struc),
          c$level byte,
          ex$val word,
          eoi literally '20H';

317 2      IO$534$base=IO$base$addr;
318 2      int$534$level=int$level;
319 2      ret$data$p=g$ret$data$p;
320 2      call rq$set$interrupt(
          /* level */      int$534$level,
          /* flags */      1,
          /* entry point */ interrupt$ptr(int$534$hnd),
          /* data segment */ 0,
          /* status ptr */  @ex$val);

```

Module 7, continued

```

321 2      do forever;
322 3          call rq$wait$interrupt(int$534$level,@ex$val);
323 3          output(IO$534$base+8)=0CH;
324 3          c$level=input(IO$534$base+8) and 07H;
325 3          call rq$send$units(ret$data.int$sema(c$level),1,@ex$val);
326 3          output(IO$534$base+8)=EOI;
327 3          end; /* of do forever */
328 2      end; /* of procedure */
*****
329 1      end interrupt$534$module;

```

MODULE INFORMATION:

```

CODE AREA SIZE = 00B5H 181D
CONSTANT AREA SIZE = 0000H 0D
VARIABLE AREA SIZE = 0005H 5D
MAXIMUM STACK SIZE = 0026H 38D
541 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-86 COMPILATION

```

/* check for non-existence of mailbox. If last device has been detached
the mailbox will be deleted in this case, delete itself */

```


Module 8

```

ISIS-II PL/M-86 X167 COMPILATION OF MODULE IO534TASKMODULE
OBJECT MODULE PLACED IN :F1:iotask.OBJ
COMPILER INVOKED BY: plm86 :F1:iotask.plm PRINT(:F5:IOTASK.LST)
DEBUG COMPACT OPTIMIZE(2) ROM DATE(4/25/80)

1      io$534$task$module:
      do;

      /*****
      IO$534$TASK: TASK.

      This task receives IORS segments from the queue$io
      procedure and performs the necessary input or
      output operations on the iSBC 534 board.

      *****/

      $include(:f4:common.lit)
      = $SAVE NOLIST
      $include(:f1:point.r.ext)
      = /* external declaration of pointerize procedure */
      = $save nolist
      $include(:f4:nucprm.ext)
      = $SAVE NOLIST
      $include(:f4:nerror.lit)
      =
      = $SAVE NOLIST
      $include(:f1:rettda.lit)
      = /* literal declaration of ret$data structure for init$534$io */
      = $save nolist
      $include(:f1:iors.lit)
      = /* literal declaration for iors */
      = $save nolist

314 1      declare
          begin$io$task$data byte public,
          req$mbox$token external,
          f$detach$device literally '5',
          f$read literally '0',
          f$write literally '1';

315 1      IO$534$task: procedure reentrant public;

316 2      declare
          iors$token,
          iors$sp pointer,
          iors based iors$sp IO$request$result$segment,
          ex$val word,
          resp$token,
          buff$sp pointer,
          buf based buff$sp (1) byte,
          i word,
          unit byte,
          ret$data$sp pointer,
          ret$data based ret$data$sp structure(ret$data$struc),
          c$val word;

317 2      do forever;
318 3          iors$token=req$receive$message(req$mbox$token,0FFFFH,@resp$token,@ex$val);

      /* check for non-existence of mailbox. IF last device has been detached
      the mailbox will be deleted In this case, delete thyself */

319 3          if ex$val= E$exist then
320 3              call rq$delete$task(0,@ex$val);
321 3          iors$sp=pointerize(iors$token);

```

Module 8, continued

```

322 3      buff$P=iors.buff$P;
323 3      unit=iors.unit;
324 3      iors.actual=0;
325 3      i=0;
326 3      ret$data$P=iors.aux$P;

327 3      if iors.funct = f$detach$device then
328 3          do;
329 4              call rq$send$message(
/* mbox token */      resp$t,
/* object token */    iors$t,
/* response token */   0,
/* status ptr */       @ex$Sval);
330 4              call rq$delete$task(0,@ex$Sval);
331 4              end;

332 3      if iors.funct=f$read then
333 3          do while iors.count > 0;
334 4              c$Sval=rq$receive$units(
/* sema */      ret$data.int$sema(2*unit),
/* units */    1,
/* time */     0FFFFH,
/* status */   @ex$Sval);

335 4              buf(i)=input(ret$data.usart$data$port(unit)) and 07FH;
336 4              i=i+1;
337 4              iors.count=iors.count-1;
338 4              iors.actual=iors.actual+1;
339 4              end;

340 3      else if iors.funct=f$write then
341 3          do while iors.count > 0;
342 4              c$Sval=rq$receive$units(
/* sema */      ret$data.int$sema(2*unit+1),
/* units */    1,
/* time */     0FFFFH,
/* status */   @ex$Sval);
343 4              output(ret$data.usart$data$port(unit))=buf(i);
344 4              i=i+1;
345 4              iors.count=iors.count-1;
346 4              iors.actual=iors.actual+1;
347 4              end;

349 3      iors.status=E$OK;
350 3      iors.done=TRUE;
351 3      call rq$send$message(iors.resp$mbox,iors$t,0,@ex$Sval);
352 2      end; /* of procedure */
353 1      end io$534$task$module;

```

MODULE INFORMATION:

CODE AREA SIZE	= 018DH	397D
CONSTANT AREA SIZE	= 000H	0D
VARIABLE AREA SIZE	= 0001H	1D
MAXIMUM STACK SIZE	= 0028H	40D
624 LINES READ		
0 PROGRAM ERROR(S)		

END OF PL/M-86 COMPILATION

CODE AREA SIZE	= 000H
CONSTANT AREA SIZE	= 000H
VARIABLE AREA SIZE	= 000H
MAXIMUM STACK SIZE	= 000H
624 LINES READ	
0 PROGRAM ERROR(S)	

END OF PL/M-86 COMPILATION

Module 9

ISIS-II PL/M-86 X167 COMPILATION OF MODULE INIT534HW.
 OBJECT MODULE PLACED IN :F1:inithw.OBJ
 COMPILER INVOKED BY: plm86 :F1:inithw.plm PRINT(:F5:INITHW.LST)
 DEBUG COMPACT OPTIMIZE(2) ROM DATE(4/25/80)

```

1      init$534$hw:
      do;

      /*****
      init$534$hw: PUBLIC PROCEDURE.

      This procedure initializes the iSBC 534 hardware and
      sets up the device dependent fields of the ret$data
      segment which will be used by the queue$io procedures.

      *****/

      $include(:f4:common.lit)
      = $SAVE NOLIST
      $include(:f1:ret$dtl.lit)
      = /* literal declaration of ret$data structure for init$534$io */
      = $save nolist

12 1  and init$534$hw: procedure(ret$data$sp) reentrant public;

13 2      declare
          ret$data$sp pointer,
          ret$data based ret$data$sp structure(ret$data$struct),
          (base,i) byte;

14 2      base=ret$data.io$base;
15 2      output(base+0FH)=0; /* board reset */
16 2      output(base+0DH)=0; /* select data block */
17 2      output(base+8)=16H; /* output ICW1 */
18 2      output(base+9)=0; /* output ICW2 */
19 2      output(base+9)=0; /* output mask word */

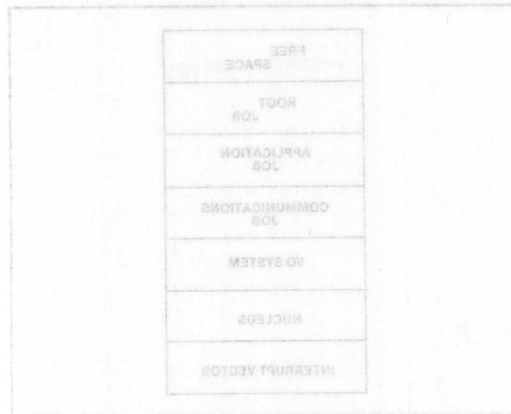
      /* attach$device calls will initialize usarts and timers */
      /* set up tables of port addresses for use by queue$io procs */

20 2      ret$data.timer$cmd(0),ret$data.timer$cmd(3)=36H;
21 2      ret$data.timer$cmd(1)=76H;
22 2      ret$data.timer$cmd(2)=0B6H;
23 2      do i=0 to 3;
24 3          ret$data.usart$cmd$port(i)=base+2*i+1;
25 3          ret$data.usart$data$port(i)=base+2*i;
26 3          ret$data.timer$load$port(i)=base+i;
27 3      end;
28 2      ret$data.timer$load$port(3)=base+4;
29 2      ret$data.timer$cmd$port(0),
          ret$data.timer$cmd$port(1),
          ret$data.timer$cmd$port(2)=base+3;
30 2      ret$data.timer$cmd$port(3)=base+7;
31 2      return;
32 2      end;
33 1      end init$534$hw;
  
```

MODULE INFORMATION:

CODE AREA SIZE	= 00E4H	228D
CONSTANT AREA SIZE	= 0000H	0D
VARIABLE AREA SIZE	= 0000H	0D
MAXIMUM STACK SIZE	= 0008H	8D
77 LINES READ		
0 PROGRAM ERROR(S)		

END OF PL/M-86 COMPILATION



System Memory Map

APPENDIX B

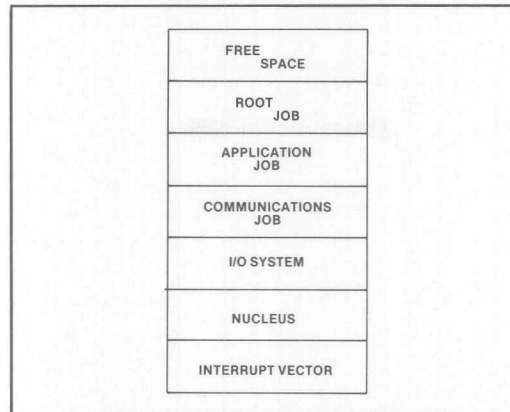
Configuration Listings/Worksheets

```

1 ***** NUCLOC.CSD *****
2
3 THIS SUBMIT FILE LINKS THE NUCLEUS.
4
5 *****
6
7 *****
8
9 *****
10
11 *****
12
13 *****
14
15 *****
16
17 *****
18
19 *****
20
21 *****
22
23 *****
24
25 *****
26
27 *****
28
29 *****
30
31 *****
32
33 *****
34
35 *****
36
37 *****
38
39 *****
40
41 *****
42
43 *****
44
45 *****
46
47 *****
48
49 *****
50
51 *****
52
53 *****
54
55 *****
56
57 *****
58
59 *****
60
61 *****
62
63 *****
64
65 *****
66
67 *****
68
69 *****
70
71 *****
72
73 *****
74
75 *****
76
77 *****
78
79 *****
80
81 *****
82
83 *****
84
85 *****
86
87 *****
88
89 *****
90
91 *****
92
93 *****
94
95 *****
96
97 *****
98
99 *****
100

```

Nucleus Link and Locate Commands



System Memory Map

```

;
;  ***----- NUCLNK.CSD -----***
;
;THIS SUBMIT FILE LINKS THE NUCLEUS.
;
:F0:LINK86 &
:F1:NUC86.LIB(NENTRY), &
:F1:NUC86.LIB &
TO :F1:NUCLUS.LNK MAP PRINT(:F1:NUCLUS.MP1) NAME(NUCLEUS)
;
;
;  ***----- NUCLOC.CSD -----***
;
;THIS SUBMIT FILE LOCATES THE NUCLEUS IN MEMORY.
;
;
:F0:LOC86 &
:F1:NUCLUS.LNK TO :F1:NUCLUS MAP PRINT(:F1:NUCLUS.MP2) SC(3) &
RESERVE(0 TO 7FFH) SEGSIZE(STACK(0)) &
ORDER(CLASSES(CODE,DATA,STACK,MEMORY)) &
OBJECTCONTROLS(NOLINES,NO COMMENTS,NOPUBLICS,NOSYMBOLS)

```

Nucleus Link and Locate Commands


```

; ios(date,origin)
; Sample I/O System .csd file to link and locate an I/O System.
; This file links an I/O System with the timer included.
; This .csd file assumes the I/O System configuration module is
; iocnfg.a86 (found on the release diskette).
;
; The origin parameter sets the low address of the I/O System;
; all the segments are contiguous in memory.
asm86 :fl:iocnfg.a86 date(%0)
link86 &
:fl:ios.lib(iocinit), &
:fl:iocnfg.obj, &
:fl:ios.lib, &
:fl:rpifc.lib &
to :fl:ios.lnk map print(:fl:ios.mp1)
loc86 :fl:ios.lnk to :fl:ios map sc(3) print(:fl:ios.mp2) &
oc(noli,nopl,nocm,nosb) &
order(classes(code,data,stack,memory)) &
addresses(classes(code(%1))) &
segsize(stack(0))

```

I/O System Link and Locate Commands

```

; Submit file to generate located version of file transaction job
;
link86 &
:fl:ftinit.obj, &
:fl:listen.obj, &
:fl:worker.obj, &
:fl:ptrintr.obj, &
:fl:rpifc.lib &
to :fl:apex1.lnk map print(:fl:apex1.mp1)
loc86 :fl:apex1.lnk to :fl:apex1 map sc(3) print(:fl:apex1.mp2) &
oc(noli,nopl,nocm,nosb) &
order(classes(code,data,stack,memory)) &
addresses(classes(code(%1))) &
segsize(stack(0))

```

File Transaction Job; Link and Locate Commands

```

;
; Submit file to generate located version of communications job
;
link86 &
:fl:cminit.obj, &
:fl:comm.lib, &
:fl:ptrintr.obj, &
:fl:rpifc.lib &
to :fl:comm.lnk map print(:fl:apex1.mp1)
loc86 :fl:comm.lnk to :fl:comm map sc(3) print(:fl:comm.mp2) &
oc(noli,nopl,nocm,nosb) &
order(classes(code,data,stack,memory)) &
addresses(classes(code(%1))) &
segsize(stack(0))

```

Communications Job; Link and Locate Commands

```

077EH 10E4H PUB INITDEVICETABLES      077EH 0FBCH PUB NAMEDDELETE
077EH 0EB3H PUB DECRUSECOUNT          077EH 0E51H PUB UNLINKCONN
077EH 0CA8H PUB NAMEDCHANGEACCES       077EH 0B5AH PUB ATTACHNAMEDFILE
→ 077EH 073EH PUB ATTACHDEVICETASK      077EH 0574H PUB ILLEGALFUNCT
077EH 003EH PUB RQAIOSINITTASK         077EH 0006H PUB COPYRIGHT

```

SEGMENT MAP

START	STOP	LENGTH	ALIGN	NAME	CLASS
077E0H	1453EH	CD5FH	W	CODE	CODE
14540H	145FFH	00C0H	W	REQ_TABLE	CODE
14600H	146DFH	00E0H	W	IOS_TABLE	CODE
→ 146E0H	14745H	0066H	W	DATA	DATA
14746H	14746H	0000H	W	STACK	STACK
14750H	14750H	0000H	G	??SEG	
→ 14750H	14750H	0000H	W	MEMORY	MEMORY

Locate Map for I/O System

(The "→" indicates entries for job macros and memory map)

```

1475H 079EH PUB SETUP544                1475H 06C5H PUB PACKETINPUT
1475H 05B5H PUB INDEX                    → 1475H 0572H PUB COMMINTTASKENTRY

```

SEGMENT MAP

START	STOP	LENGTH	ALIGN	NAME	CLASS
→ 14750H	15BCDH	147DH	W	CODE	CODE
→ 15BD0H	170D2H	1502H	W	DATA	DATA
170D2H	1712EH	004CH	W	STACK	STACK
17130H	17130H	0000H	G	??SEG	
→ 17130H	17130H	0000H	W	MEMORY	MEMORY

Locate Map for Communications Job

```

→ 17D6H 03B5H PUB BEGINLISTENERTASKDATA 1713H 0153H PUB POINTERIZE
1713H 0112H PUB INITTASKENTRY           1713H 0401H PUB WORKERTASK

```

SEGMENT MAP

START	STOP	LENGTH	ALIGN	NAME	CLASS
17130H	17D59H	0C29H	W	CODE	CODE
17D60H	17E28H	00C8H	W	DATA	DATA
17E30H	17E9AH	006AH	W	STACK	STACK

Locate Map for File Transaction Job

Macro call: SYSTEM (system parameters)Number of calls required: exactly oneCONFIGURATION FILE NAME APEXT

FORMAT:

	<u>parameter</u>	<u>type</u>	<u>suggested default</u>	<u>value</u>
%SYSTEM	(nucleus__entry,	base		<u>80:0</u>
	rod__size,	word	(0)	<u>10</u>
	min__trans__size,	work	(64)	<u>64</u>
	debugger,	see note	(A)	<u>N</u>
	default__e__h__provided,	see note	(N)	<u>N</u>
	mode)	2 word		<u>1</u>

NOTES:

- Valid entries for the debugger parameter include:

A Debugger available
 N No debugger available

- Valid entries for the default__e__h__provided parameter include:

Y Yes
 D Debugger
 N No

%SYSTEM Macro Worksheet

Macro call: _____																	
Macro call: SAB (for system address blocks)																	
Number of calls required: _____																	
Number of calls required: one or more																	
CONFIGURATION FILE NAME: _____																	
CONFIGURATION FILE NAME: APEX1																	
FORMAT: _____																	
FORMAT: %SAB																	
FORMAT:	<table border="1" style="width: 100%; border-collapse: collapse;"><thead><tr><th style="width: 20%;">parameter</th><th style="width: 20%;">type</th><th style="width: 20%;">suggested default</th><th style="width: 40%;">value</th></tr></thead><tbody><tr><td>start_base</td><td>base</td><td></td><td>0</td></tr><tr><td>end_base</td><td>base</td><td></td><td>1900</td></tr><tr><td>type</td><td>word</td><td></td><td>U</td></tr></tbody></table>	parameter	type	suggested default	value	start_base	base		0	end_base	base		1900	type	word		U
parameter	type	suggested default	value														
start_base	base		0														
end_base	base		1900														
type	word		U														
NOTES: _____																	
NOTES:																	
<ol style="list-style-type: none">1. The type parameter is reserved for future use. Enter the character U for this parameter.2. A SAB is declared between start_base:0 and end_base:F, inclusive.																	

%SAB Macro Worksheet

%SYSTEM Macro Worksheet

Macro call: JOB (defines first-level jobs)

Number of calls required: one for each first-level job

CONFIGURATION FILE NAME: APEX 1

FORMAT:

	<u>parameter</u>	<u>suggested type</u>	<u>default</u>	<u>value</u>
%JOB	(directory_size,	word	(0)	<u>0</u>
	pool_min,	word		<u>OFFFH</u>
	pool_max,	word	(OFFFH)	<u>OFFFH</u>
	max_objects,	word		<u>FFFF</u>
	max_tasks,	word		<u>FFFF</u>
	max_job_priority,	byte		<u>129</u>
	exception_handler_entry,	addr	(0:0)	<u>0:0</u>
	exception_handler_mode,	byte	(1)	<u>1</u>
	job_flags,	word	(0)	<u>0</u>
	init_task_priority,	byte		<u>1713:112</u>
	data_segment_base,	base	(0)	<u>17D6</u>
	stack_pointer,	addr	(0:0)	<u>0:0</u>
	stack_size,	word	(512)	<u>512</u>
	task_flags)	word	(0)	<u>0</u>

NOTE:

1. addr is specified as base:offset

%JOB Macro Worksheet


```

%sub(0,1900,U)
%job(0,300h,0FFFh,0ffffh,0ffffh,0,0:0,0,0,128,77e:3e,146e,0:0,512,0)
%job(0,1FFH,0FFFFH,0FFFFH,0FFFFH,128,0:0,0,0,131,1475:572,15bd,0:0,400,0)
%job(0,300H,0FFFFH,0FFFFH,0FFFFH,128,0:0,1,0,130,1713:112,17d6,0:0,400H,0)
%system(80,10,64,N,N,1)

```

Configuration File Apex 1.CNF

```

;
;*****CTABLE.CSD*****
;
; SUBMIT :Fx:CTABLE( fsys, fin, fout, config_file, date )
;
; This submit file assembles the CTABLE module, where:
;   fsys      = the system disk containing ASM86
;   fin       = the source/input disk (F1 is assumed)
;   fout      = the object/listing/output disk
;   config_file = the path-name of the configuration file
;   date      = the date
;
copy %3 to :f1:config.cnf b
;
%0:asm86 :%1:ctable.a86 pr(:%2:ctable.lst) oj(:%2:ctable.obj) date(%4) &
xref debug ep

```

Submit File to Generate Configuration Table

```

;
;*****CLNKRJ.CSD*****
;
; SUBMIT :Fx:CLNKRJ( fsys, fin, fout )
;
; This submit file links the Root-Job, where:
;   fsys      = the system disk containing LINK86
;   fin       = the source/input disk
;   fout      = the object/listing/output disk
;
;%0:link86 :%1:croot.lib(root),&
;           :%2:ctable.obj,&
;           :%1:croot.lib &
to :%2:rootjb.lnk map pr(:%2:rootjb.mpl)
;

```

Submit File to Link the Root Job

```

;
;***-***-***-***-***-***- CLOCRJ.CSD ---***-***-***-***-***
;
; SUBMIT :Fx:CLOCRJ( fsys, fin, fout )
;
; This submit file locates the Root-Job, where:
;   fsys = the system disk containing LOC86
;   fin  = source/input disk
;   fout = object/listing/output disk
;
;-- NOTE: BE SURE TO REPLACE THE "?????" BELOW WITH THE APPROPRIATE
;-- ADDRESS THE ROOT-JOB IS TO BE LOCATED AT!!
;
:%0:loc86 :%2:rootjb.lnk to :%2:rootjb &
map pr(:%2:rootjb.mp2) sc(3) &
name(ROOT_JOB) oc(nocm,noli,nopl,nosb) &
segsz(stack(200h)) &
order(classes(data,stack,memory,code)) &
addresses(classes(data(12C00H)))

```

Submit File to Locate Root Job



APPLICATION NOTE

AP-88

May 1980

Multiprocessing Extensions for the RMX/80™ Real-Time Executive

Peter Andersen
OEM Microcomputer Systems Applications

INTRODUCTION	2-133
Multiprocessor Strategies	2-133
Process Example	2-133
MULTIPROCESSOR MULTITASKING	
OPERATIONS	2-135
Bus Priority Resolution Lines	2-135
Resource Contention	2-136
Bus Lock	2-136
Semaphores	2-137
Hardware Semaphores	2-137
Software Semaphores	2-138

MULTIPROCESSOR COMMUNICATION	2-138
Messages	2-138
Exchanges	2-138
RMX/80™ Message Communications	2-140
Multiprocessor Message Transfer	2-140
Waiting for a Message	2-140
Sending a Message	2-142
Responding to a Request	2-144
Testing an Exchange	2-144
Initialization of Triggers	2-144
Use of Semaphores	2-144
System Resource Sharing	2-145
APPLICATION EXAMPLES	2-146
Supervisor Implementation	2-146
Waiting Facilities	2-147
Other Application Tasks	2-148
Total Application Configuration	2-148
CONCLUSION	2-148
APPENDIX A — Multiprocessor Message Transfer Primitives	2-152
APPENDIX B — Bit Program Listing	2-152

Using the RMX/80™ Nucleus in Multiprocessor Applications

Contents

INTRODUCTION	2-133
Multiprocessor Strengths	2-133
Process Example	2-133

MULTIBUS™ MULTITASKING OPERATIONS	2-135
Bus Priority Resolution Lines	2-135
Resource Contention	2-136
Bus Lock	2-136
Semaphores	2-137
Hardware Semaphores	2-137
Software Semaphores	2-138

MULTIPROCESSOR COMMUNICATIONS	2-138
Messages	2-139
Exchanges	2-139
RMX/80™ Nucleus Communications ...	2-140
Multiprocessor Message Transfers	2-140
Waiting for a Message	2-140
Sending a Message	2-142
Responding to Interrupt Request ...	2-144
Testing an Exchange	2-144
Initialization of Descriptors	2-144
Use of Semaphores	2-144
System Resource Sharing	2-145

APPLICATION EXAMPLE	2-146
Supervisor Implementation	2-146
Weighing Ingredients	2-147
Other Application Tasks	2-149
Total Application Configuration	2-149

CONCLUSION	2-149
-------------------------	--------------

APPENDIX A — Multiprocessor Message Transfer Programs	2-152
--	--------------

APPENDIX B — BIPI Program Listing ...	2-168
--	--------------

I. INTRODUCTION

As computer systems become more complex, a recurring need occurs to use multiple processing units. Multiprocessing can, in many applications, increase throughput, enhance reliability, or create a more modular design approach. Intel's single board computer family provides a convenient tool for creating a multiprocessing environment when combined with the Multibus system bus architecture. The use of the Intel RMX/80 Real-Time Multitasking Executive simplifies the creation of modular designed system architectures. It also allows more efficient use of the processor's capabilities by sharing them between several independent tasks.

This application note provides insight as to the techniques which can be used to create a multiprocessing system which is fully compatible with the use of Intel's RMX/80 multitasking executive. Both hardware and software capabilities of the Intel single board computers are discussed. For example, an explanation of both serial and parallel priority resolution techniques for creating a multimaster system are included. The techniques for using the bus lock features to provide mutual exclusion of resources and for creating both hardware and software semaphores is also a part of the application note. Many of the multiprocessing principles are combined to create the capability of transferring messages between tasks operating on different processor boards, thus creating a combined multiprocessor/multitasking operating system.

A potentially complex application is examined and the benefits gained by using a multiprocessing approach are discussed. The various requirements defined from the application are used to demonstrate the available features of the Intel product line as applied to multiprocessing.

Extensions of the features are developed where necessary to create a complete solution to the design exercise. For example, a capability is provided for a flexible operator interface to the system which provides certain global functions to the multiple processors. This capability, called the Common System Module (CSM), includes the required drivers for disk drives and for an operator's CRT terminal. The operator interface includes a BASIC interpreter capability.

Finally, the application is modularized and the techniques which make the control solution easily implemented using multiprocessing techniques are demonstrated.

It is assumed that the reader has a fundamental knowledge of multitasking operating system concepts and is familiar with the use of PL/M-80 as a language on Intel's single board computers. In addition, a working knowledge of BASIC is assumed. The reader not having these skills should refer to the documents referenced in the Related Publications section of this application note.

Multiprocessing Strengths

The most common use of multiprocessing is to offload a main processor of low level duties in order to increase system throughput. Significant examples of this technique are the use of Intel UPI devices such as the 8741A and the use of intelligent slave boards such as the Intel iSBC 544 Intelligent Communications Controller or the iSBC 569 Intelligent Digital Controller. The techniques which are developed in this application note do not apply to these multiprocessor implementations because they have been extensively covered in other application notes.

Processors can be offloaded of other than low level functions in complex systems. Just as a real-time application can be divided into functional tasks, these tasks can be grouped according to function. Each function or class of tasks can be assigned to a single board computer. The system throughput can thus be increased since tasks can now run concurrently and, using the software procedures developed in this application note, can communicate among themselves as though they were operating on the same processor board. Single board computers operating in this manner are known as multimaster computers.

A second instance where multiprocessing can be considered in a system design is in those cases where the operational integrity can be enhanced by having more than one processor. In these cases, the application may require that no single component failure can completely cause the system to fail. Multiprocessing provides this feature by allowing control of those operations associated with a processor board to continue to function even when another board has failed. In many cases, provision can be made to have the remaining board or boards take over some or all of the functions which were associated with the failed board. Even though this would result in slower throughput, critical functions would still be performed by the computer system.

Another benefit is the modular system design and expansion capabilities which can be realized by using multiprocessing in industrial control applications. Here, it has been found that process definitions and operations tend to be rather dynamic. As products are improved or new ones added to the operation, the control system must be capable of being updated to conform to the new product flow. It is common to have an added requirement that the changes in a process must be made with no impact on other processes being controlled by the same system. Using a multimaster board for each process allows this operation to be easily implemented.

Process Example

To emphasize the advantages which can be gained using multiprocessing, consider the chemical production facility whose process flow is shown in Figure 1. This product flow chart is typical of many found in the chemical industry where a product is manufactured from several

raw ingredients which are purchased. This type of product lends itself well to automation. Consider, then, the approach to designing a control system which will operate in the facility which has been defined.

The first design goal is to define global system tasks which need to be performed by the system. Examination of the flow diagram in Figure 1 might lead one to define the system tasks as:

- 1) Inventory control
- 2) Quality control
- 3) Production scheduling
- 4) Weighing ingredients
- 5) Mixing ingredients
- 6) Drying and forming product
- 7) Packaging

The choice just made certainly is not the only valid possibility but it should be adequate to allow the design discussion to continue.

The system tasks listed can be grouped according to supervisory and control tasks. Items 1 through 3 fall into the supervisory category and items 4 through 7 belong to control category. This breakdown also groups

the items according to the type of programs which will be required to implement the functions.

The grouping of system functions into categories suggests that a multimaster approach is a good design path to create a system solution. Five multimaster boards are suggested, one for the supervisor and four boards which provide the control capabilities. This functional grouping is seen in Figure 2.

The examination of the product flow and system interactions shown in Figure 1 indicate that extensive communications are required between the various boards of the system. In addition, several tasks are required on each multimaster board and communications are required between these tasks.

Intel's RMX/80 Real-Time Multitasking Executive provides a simple method of handling the on-board communication operations by supporting the transmission and receipt of messages for data communication and synchronization. After a discussion of the hardware features which support multiprocessing, the application note will deal with the extension of the RMX/80 nucleus to support the transmission of messages between tasks which reside on different single board computers.

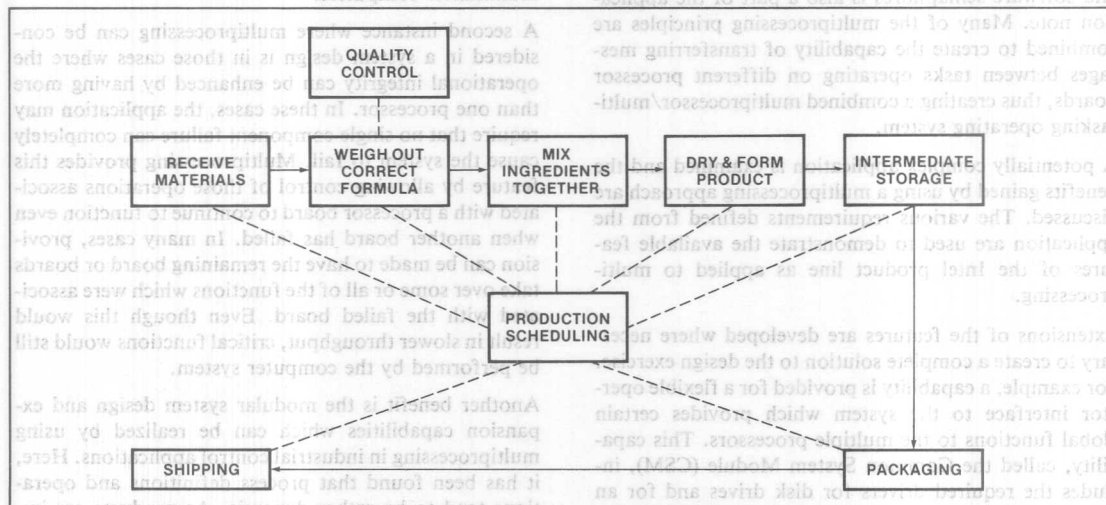


Figure 1. Chemical Production Flow

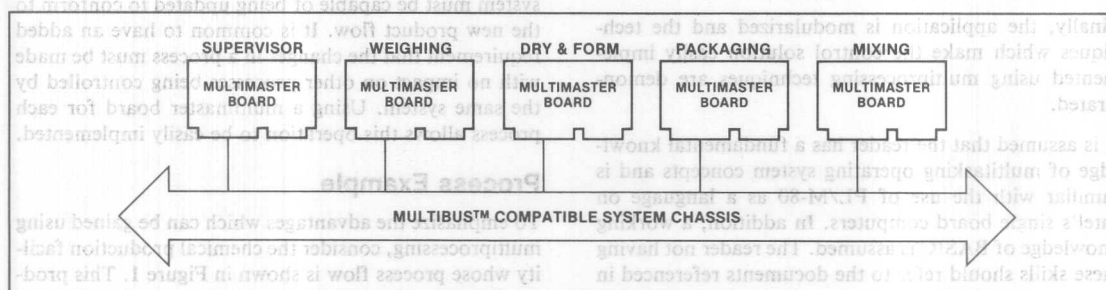


Figure 2. Multimaster System Solution

II. MULTIBUS™ MULTITASKING OPERATIONS

The interactions of the various multimaster board functions occur via the medium of the Multibus system bus. This bus structure is designed to easily support the use of more than one master processor in a system. Subsequent paragraphs explain in some detail the features of the Multibus system bus which allow multimaster operations when used with multimaster compatible single board computers.

Bus Priority Resolution Lines

The Multibus system bus uses seven control lines to support the bus priority resolution techniques. Before continuing with a discussion of these techniques, the reader should be provided with a definition of these lines and their functions. These lines are defined in subsequent paragraphs. One line is used for the system clock. It is:

BCLK/ Bus Clock; the negative edge of BCLK/ is used to synchronize bus priority resolution circuits. BCLK/ is asynchronous to the CPU clock. It has a 100 ns minimum period and a 35 to 65 percent duty cycle.

All Intel single board computers which are capable of being used as a bus master can provide this clock signal. In a multiple processor application, it is necessary to select only one board which is to actually supply this signal to the system bus. On all other processor boards, this signal must be disabled by removing the jumpers as indicated in the appropriate Hardware Reference Manual (HRM).

Several bus signals are dedicated to providing information as to the status of the bus to processors which are attempting to obtain control for their own data transfer. These are defined to be:

BPRN/ Bus Priority In Signal; indicates to a requesting bus master board that no higher priority board is requesting use of the system bus. BPRN/ is synchronized with BCLK/. The signal is not based on the backplane and must be generated and connected by the user as will be seen.

BUSY/ Bus Busy Signal; an open collector line driven by the current bus master to indicate that the bus is currently in use. BUSY/ prevents all other potential masters from gaining control of the bus. BUSY/ is synchronized with BCLK/.

CBRQ/ Common Bus Request; an open-collector line which is driven by all potential bus masters and is used to inform the current bus master that another device wishes to use the bus. If CBRQ/ is high, it indicates to the current bus master that no other master is requesting the bus, and therefore, the present master can retain the bus control. This saves the bus ex-

change overhead which would be required to release and again re-acquire control of the bus.

The timing relationship of these signals can be seen in Figure 3. Note that some type of bus request signal is generated by the master wishing to assert control of the system bus. Through some type of logic, a determination must be made as to the effect that the requesting master has the highest priority. If so, the logic informs the requesting master by setting the BPRN/ signal low.

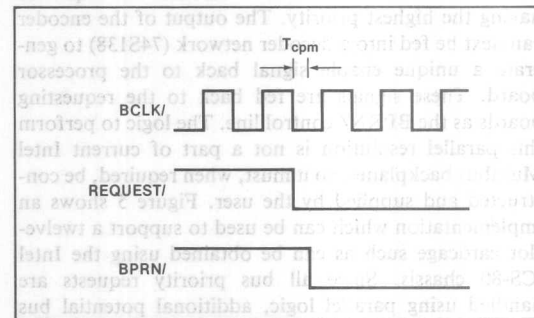


Figure 3. Bus Master Request Timing

Two signals are generated by single board products designed to be bus masters. These signals are used to generate the request signal. The request lines provided are known as:

BPRO/ Bus Priority Out Signal; used with serial bus priority resolution schemes. BPRO/ is passed to the BPRN/ input of the master module with the next lower priority. BPRO/ is synchronized with BCLK/ and is not based on the backplane.

BREQ/ Bus Request Signal; used with a parallel bus priority network to indicate that a particular master board requires the use of the bus for one or more data transfers. BREQ/ is synchronized with BCLK/. It is not based on the backplane.

The most easily implemented technique for supporting bus priority resolution is the serial or the daisy chain method. In this method, each board examines its BPRN/ line. If the line is low and the master desires to use the system bus, it may do so. Internal board logic then places the BPRO/ line high to inhibit boards along the daisy chain with lower priority from gaining control. Any board in the chain with its BPRN/ line high or which is requesting priority will set the BPRO/ line high. The wiring technique and logic of the serial priority resolution scheme is shown in Figure 4. As long as the worst case propagation delay time between the highest and the lowest potential bus master does not exceed 30 ns, the technique provides a simple and effective method for resolving bus contention problems. Because of logic delays on iSBC boards, it has been found that the serial technique can only be used for a maximum of three bus masters when using a 100 ns clock rate. If

more masters are desired, either the clock must be slowed or the priority resolution technique must be modified. Thus, the BREQ/ line is used to support the second or parallel priority scheme.

In a parallel priority network, all requests from bus masters are supported in parallel logic. Each master desiring use of the system bus places its BREQ/ line to a logical low condition. A parallel priority encoder chip such as a 74148 can be used to indicate the requestor having the highest priority. The output of the encoder can next be fed into a decoder network (74S138) to generate a unique enable signal back to the processor board. These signals are fed back to the requesting boards as the BPRN/ control line. The logic to perform this parallel resolution is not a part of current Intel Multibus backplanes, so it must, when required, be constructed and supplied by the user. Figure 5 shows an implementation which can be used to support a twelve-slot cardcage such as can be obtained using the Intel iCS-80 chassis. Since all bus priority requests are handled using parallel logic, additional potential bus masters may be added without adding to the signal delay time. Thus, the number of requests which may be resolved is limited only by the number of available card slots on the Multibus system bus.

Resource Contention

Even though the bus priority resolution networks described above provide against two or more bus masters taking control of the bus at the same time, they do not prevent one processor from examining and modifying data at the same time that another is operating on this data (this is true because data is only transferred in 8 or 16-bit blocks and the bus may be used by another master between transfers). Some technique for inhibiting the contention of resources must be provided if true multiprocessing is to be performed. Three candidates can be considered to support this exclusion process. The techniques required to support each are explored in subsequent paragraphs.

BUS LOCK

Single board computers provide the capability of locking the system bus from access by other bus masters. This feature is provided by means of a bus lockout flip-flop which may be set or reset by writing to a dedicated I/O port on each board. This flip-flop has the effect of keeping the BUSY/ active which keeps the master in control of the bus. In this way, a master assures that it has exclusive control of a bus common resource. In fact, it has exclusive control of all bus resources even though

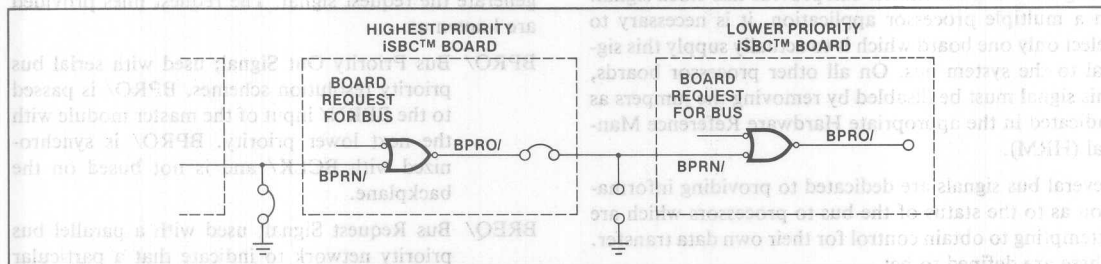


Figure 4. Serial Priority Implementation

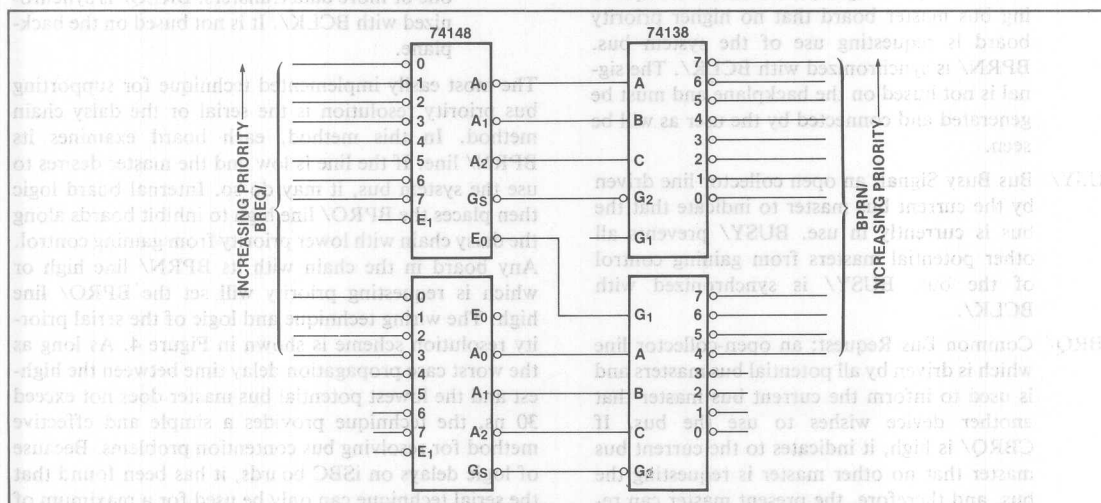


Figure 5. Parallel Priority Implementation

it may be using but one of them. This drawback prohibits other masters from using devices on the bus which are not in contention as well as those that are. In many systems, this potential delay can prohibit the use of bus locks in all but a few special applications.

SEMAPHORES

Mutual exclusion can easily be obtained by using the concept of a semaphore. A semaphore may be thought of as a type of traffic signal which can be used to prohibit access to certain roads or resources. When such a semaphore technique is used with microprocessors, it is referred to as a test-and-set operation. This means that the act of testing the semaphore not only returns the current status but also forces the semaphore to the "on" condition. If the returned status from the test indicated a previously cleared condition of the semaphore, the resource associated with it is considered available for exclusive use. Since the semaphore is now set, all other masters testing it will find a busy condition and must wait until the user processor clears the semaphore associated with the resource. Each time a processor completes its operations on a shared resource, it clears the semaphore by writing a zero to it.

In practice, two types of semaphores are commonly found, the hardware semaphore and the software semaphore. The use of each is identical and follows the general concepts outlined in the preceding paragraphs.

Hardware Semaphores

Hardware semaphores are those which are implemented using physical hardware. Logic components are used in circuits which must be designed and constructed by the

user. Currently, no Intel boards provide for this type of semaphore implementation. Hardware semaphores have the advantage over a software implementation of being faster and requiring less system bus time. The most common designs consider the semaphore as an I/O port although memory mapped designs are certainly feasible.

Hardware semaphores are conceptually thought of as a D-type flip-flop. A simplified diagram of a typical semaphore is seen in Figure 6(a). Examining the timing relationships of Figure 6(b), it is seen that, if an initial state of a clear semaphore is assumed, when a master processor reads the I/O port, a data value of "zero" will be returned. The hardware immediately sets the flip-flop to a logical "one" on the trailing edge of the read signal. Any subsequent reads will find the semaphore "set". When the user is finished with the resource, it can clear the flip-flop by performing a "write" operation to the I/O port.

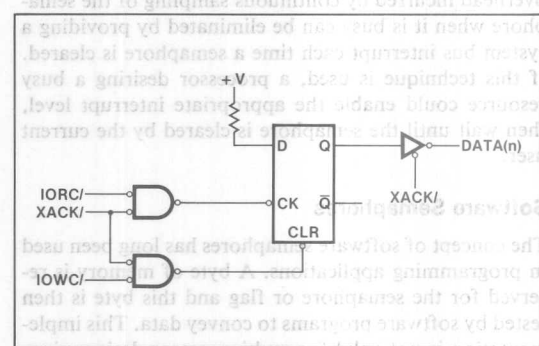


Figure 6(a). Hardware Semaphore

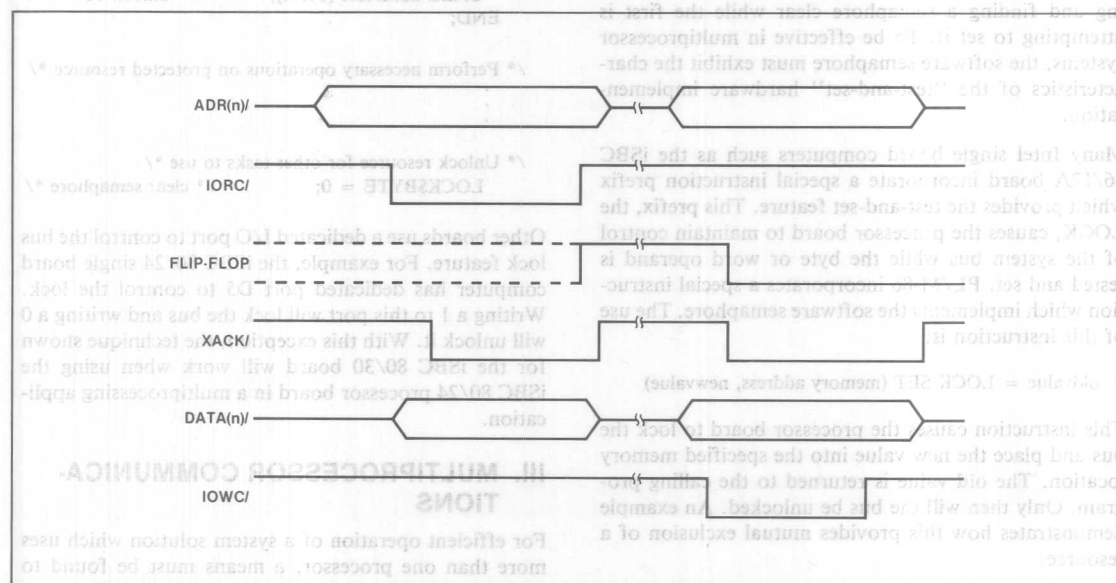


Figure 6(b). Hardware Semaphore Timing

The user can prevent contention of a system resource by incorporating a "wait for semaphore clear" into his program. For example, if a resource is protected by a semaphore at I/O port 35H, the following PL/M code could be used to protect the resource:

```
/* WAIT FOR SEMAPHORE TO CLEAR, LOCK IT */
DO WHILE INPUT (035H) < > 0; END;
/* PERFORM OPERATIONS ON RESOURCE */
/* CLEAR SEMAPHORE TO ENABLE RESOURCE */
OUTPUT (035H) = 0;
```

The hardware semaphore is seen to be an effective method of incorporating mutual exclusion in a multiprocessor application. Its major asset is that it minimizes system bus overhead and assures maximum bus throughput by higher priority master devices. Its weakness is the requirement for the user to provide hardware to implement the semaphores. In many cases, the bus overhead incurred by continuous sampling of the semaphore when it is busy can be eliminated by providing a system bus interrupt each time a semaphore is cleared. If this technique is used, a processor desiring a busy resource could enable the appropriate interrupt level, then wait until the semaphore is cleared by the current user.

Software Semaphores

The concept of software semaphores has long been used in programming applications. A byte of memory is reserved for the semaphore or flag and this byte is then tested by software programs to convey data. This implementation is not valid for multiprocessor designs since there is nothing to prevent a second processor from testing and finding a semaphore clear while the first is attempting to set it. To be effective in multiprocessor systems, the software semaphore must exhibit the characteristics of the "test-and-set" hardware implementation.

Many Intel single board computers such as the iSBC 86/12A board incorporate a special instruction prefix which provides the test-and-set feature. This prefix, the LOCK, causes the processor board to maintain control of the system bus while the byte or word operand is tested and set. PL/M-86 incorporates a special instruction which implements the software semaphore. The use of this instruction is:

```
oldvalue = LOCK SET (memory address, newvalue)
```

This instruction causes the processor board to lock the bus and place the new value into the specified memory location. The old value is returned to the calling program. Only then will the bus be unlocked. An example demonstrates how this provides mutual exclusion of a resource.

Assume that a software semaphore, LOCK\$BYTE has been declared. A value of 1 indicates that the resource is

being used and is not available. A value of 0 indicates that the common resource is available to a task. If the function reference LOCKSET(@LOCK\$BYTE, 1) is executed, the value of 1 will be assigned to LOCK\$BYTE. Furthermore, the old value of the semaphore will be returned. If a value other than 0 is returned, the statement must be repeated as the resource is being used by another task. When a value of 0 is returned as the old value, the resource has been dedicated to the calling task. When the task is finished with the shared resource, a zero must be written to the semaphore byte.

A similar technique can be used with 8-bit single board computers. For example, consider the iSBC 80/30 board. Provision has been made to provide a bus lock condition when the CPU SOD (the SOD is a output data line unique to the 8085 processor which is normally intended to be used as a serial output line) is active. A task desiring to gain exclusive access to a common resource, can lock the bus, then test and set the semaphore flag; finally, the bus can be freed by clearing the SOD signal. An example will make this more clear.

Consider the same semaphore which was used in the PL/M-86 discussion. A task executing on an iSBC 80/30 board and desiring to use the protected resource would have code similar to the following:

```
/* WAIT FOR RESOURCE AVAILABLE */
OLDVALUE = 1;
DO WHILE OLDVALUE = 1;
CALL S$MASK (0C0H); /* lock bus */
OLD VALUE = LOCK$BYTE; /* get old value */
LOCK$BYTE = 1; /* set semaphore */
CALL S$MASK (040H); /* unlock bus */
END;

/* Perform necessary operations on protected resource */
:

/* Unlock resource for other tasks to use */
LOCK$BYTE = 0; /* clear semaphore */
```

Other boards use a dedicated I/O port to control the bus lock feature. For example, the iSBC 80/24 single board computer has dedicated port D5 to control the lock. Writing a 1 to this port will lock the bus and writing a 0 will unlock it. With this exception, the technique shown for the iSBC 80/30 board will work when using the iSBC 80/24 processor board in a multiprocessing application.

III. MULTIPROCESSOR COMMUNICATIONS

For efficient operation of a system solution which uses more than one processor, a means must be found to transmit data between the system bus master boards. The data, so transferred, might be used to pass param-

eters, to move data strings, or to synchronize events on the various boards. While it is not the intent of this application note to fully explain all possible techniques, it does provide some insight into the most common multiprocessor data transfer methods.

After a short discussion of various transfer mechanisms, the techniques used in the RMX/80 nucleus are described. The descriptions provide a basis for the development of an extension to the nucleus which allows tasks to reside on multiple boards in a multi-master environment.

The most straightforward technique to communicate between processors is the use of flags or semaphores to synchronize operations. The procedures used in this type of data transfer have been explained in previous paragraphs. As will be seen, this technique will be later combined with more complex methods to synchronize the transfer of data.

A second type of communications which may be used involves creating and maintaining data queues. This method was created to support a special case of multiprocessing, the use of intelligent slaves. The creation and support of data queues has been thoroughly detailed in an Intel application note, *AP-53, Using the iSBC 544 Intelligent Communications Controller*. It should be noted that, even though created for intelligent slaves, the technique is certainly applicable to the general case of multiprocessor communications.

The generalized data queue is designed to primarily deal with the movement of data strings between processors. The queue provides independent operation for each of the two involved processors, which may operate upon the data contained within the queue independently. The queue handlers, however, must interact since the queue pointers must never be allowed to pass each other or invalid data would be handled. This implies that mutual exclusion must be applied to the pointers to guarantee their integrity. If this is done, the queue can readily be applied to the special cases where string data transfers must take place between various processor boards.

Multiprocessor systems infer that the application breaks down into functionally independent blocks or "tasks". Frequently, these global tasks will most often be further subdivided into smaller on-board or local tasks. These systems will frequently use real-time executive operating systems.

Communications between tasks can involve the transfer of many types of information. In some cases, data strings must be moved. In others, the transfer may be only used for synchronization of events. Intel's RMX/80 multitasking operating system supports the movement of data (messages) through the use of exchanges. In this implementation, actual messages are not moved in memory, but instead, a pointer is generated to the message area and this variable is passed between tasks. Before proceeding with the development of a multiprocessor message driver, some time must be

taken to assure that the message/exchange relationships are fully understood by the reader.

Messages

A message is a collection of data that one task sends to another task. It may be thought of in the context of a letter which is to be sent from one person (task) to another. Like a letter, the purpose of the message is to convey information or to request a service. Messages used in RMX/80 systems conform to a standard format similar to conventional letter connotations. It contains a heading and a variable length data area. The heading can be from 5 to 9 bytes in length and corresponds to the format shown in Figure 7.

The LINK PORTION of the heading is used by the RMX/80 nucleus to keep track of other messages which are addressed to the same place (exchanges). If no other messages are present, the field will be set to zero.

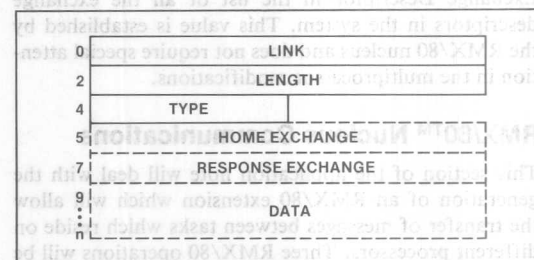


Figure 7. RMX/80™ Message Format

Exchanges

The exchange can be thought of as a mailbox into which messages are placed. In reality, only the location pointer of the message is placed into the exchange. Each exchange is defined by an Exchange Descriptor area of RAM memory. The format of an Exchange Descriptor is shown in Figure 8. The purposes of each field will be defined in the following paragraphs and should be understood since the development of a multiprocessor RMX/80 exchange protocol will use many of these fields.

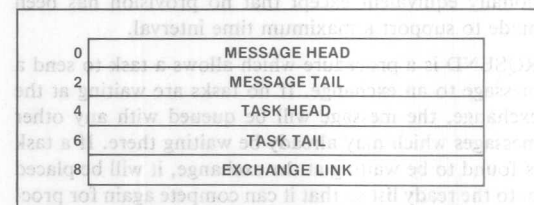


Figure 8. RMX/80™ Exchange Descriptor

MESSAGE HEAD is used to provide a pointer to the first message which has been placed into the exchange. If no messages are present at an exchange, the value of the field will be set to zero. As will be seen, this field can be tested by a task to determine if a message is already

waiting at an exchange when the task is to accept a message.

MESSAGE TAIL provides a pointer to the last message waiting at an exchange. If no messages are waiting, the pointer will be set to the address of the exchange. The multiprocessor interface which will be designed must provide an update capability for maintaining this field.

TASK HEAD is a pointer to the first task which is waiting at the exchange for a message. If no task is waiting, this field will be set to zero. Again, a simple test, when sending a message to see if a task is to be activated, is to test the field for a zero value.

TASK TAIL provides another pointer which indicates the last task which is waiting at an exchange. As with the Message Tail, the field is set to the address of the exchange when no task is waiting at the exchange.

EXCHANGE LINK contains the address of the next Exchange Descriptor in the list of all the exchange descriptors in the system. This value is established by the RMX/80 nucleus and does not require special attention in the multiprocessor modifications.

RMX/80™ Nucleus Communications

This section of the application note will deal with the generation of an RMX/80 extension which will allow the transfer of messages between tasks which reside on different processors. Three RMX/80 operations will be supported and parallel procedures developed. These will be the nucleus functions RQWAIT, RQSEND, and RQACPT.

RQWAIT is used to allow a task to receive a message from an exchange. If a message is available, its location will be transferred to the receiving task and program execution will continue. When no message is available, the task will be placed on the wait list until such time that a message has been placed into the exchange by some other task. RQWAIT provides for the capability of allowing a maximum time during which a task will wait at an exchange for a message. The multiprocessor equivalent of this function which is explained in this application note is identified as IPWAIT. It is functionally equivalent except that no provision has been made to support a maximum time interval.

RQSEND is a procedure which allows a task to send a message to an exchange. If no tasks are waiting at the exchange, the message will be queued with any other messages which may already be waiting there. If a task is found to be waiting at the exchange, it will be placed onto the ready list so that it can compete again for processor resources. The multiprocessor equivalent defined by this note is called IPSEND.

RQACPT provides a procedure which gives a task the ability to test an exchange to see if a message is present. If one is waiting, its address will be returned to the calling task. If no message is available, a value of zero is

returned. The multiprocessor equivalent is called IPACPT.

The concepts used to develop each of the three extensions are discussed in the following paragraphs. Not only should this development assist the user in creating a multiprocessor executive, but it should also help in the understanding of the operation of the RMX/80 nucleus.

Multiprocessor Message Transfers

Normal message transfers take place between tasks which are resident on the same processor board and thus are able to share a common memory area for the storage of the message's data. When a multiprocessor configuration is implemented, these tasks may not necessarily have all memory areas accessible to each other. An area of memory which is common to all processor boards must be created to implement a multiprocessor system. This memory will be hereafter referenced as GLOBAL memory. All common resources which may be needed or used by a task which might have a need to perform multiprocessor operations must use the Global memory area.

The implementation of a RMX/80 extension which provides a multiprocessor communications driver can be performed using the knowledge gained from the definitions of the fields in the messages and exchanges. The development will begin with the concepts involved in executing an exchange wait.

WAITING FOR A MESSAGE

First consider the condition in which a message is waiting at the specified exchange when the task performs a request to obtain a message from an exchange. In this case, the message can be accepted directly (and the pointer fields updated accordingly) and the task can proceed normally. The sequence of events for this type of condition can be seen in the flow chart of Figure 9. A complete listing of the code can be found in Appendix A. References will be made to lines of code which are pertinent to the discussion by using a pointer in parenthesis. Thus a reference to code at line (1.14) will indicate program 1, line 14 in the appendix.

The procedure can examine the LINK field of the message to determine if more than one message is waiting at the exchange (1.36). A zero in the field indicates that no other messages are present. The value of the LINK field should be moved into the MESSAGE HEAD field of the Exchange Descriptor (1.36). The MESSAGE TAIL field should reflect the last message. If more messages are present, the field can be left unchanged; otherwise, the field should be set to the exchange address. The address of the message is returned to the calling program, allowing it to use the data contained in the message (1.40). Note how the use of a software semaphore has been used to prevent more than one processor modifying the exchange and message descriptor data at the same time (1.24; 1.39).

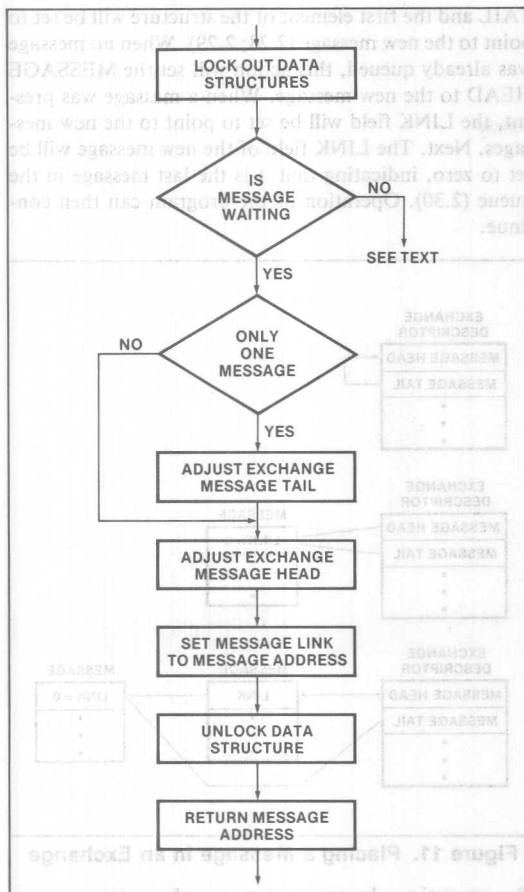


Figure 9. Getting a Message from an Exchange

The technique described above provides a direct approach to getting a message from an exchange. A more complete methodology is required when a message is not waiting at the exchange. This is the subject of the following paragraphs.

As tasks queue up at an exchange waiting for a message, provision must be made to create and maintain a link list of those tasks. Since the source code associated with RMX/80 is not normally available to the user, the effect of using the existing data structures cannot always be determined. An extension must be developed which will support the maintenance of a link list of queued tasks. Thus, each task descriptor will have a **THREAD** pointer added which will point to the next task waiting at the exchange. Each exchange descriptor will have two fields added, one for use as a **TASK HEAD** pointing to the first task waiting at the exchange, and a second field, **TASK TAIL** which points to the last task waiting at the exchange. These fields will be added to the existing RMX/80 descriptors for those tasks and descriptors which are associated with multiprocessor operations.

Figure 10 provides representations of the exchange and task descriptors as they would be configured for various conditions. Consider first the condition in which one task is already waiting at an exchange. The **TASK TAIL** and the **TASK HEAD** will be pointing to the task descriptor of the waiting task. Since only one task is waiting, the **THREAD** field of the task will contain a zero value. Appendix A contains the listing of the code required to add a task to the exchange queue. The program is the same as the one previously discussed when a message was waiting at the exchange as the calling task attempted to obtain a message. If the test indicates that no message is present (1.25), the fields must be updated

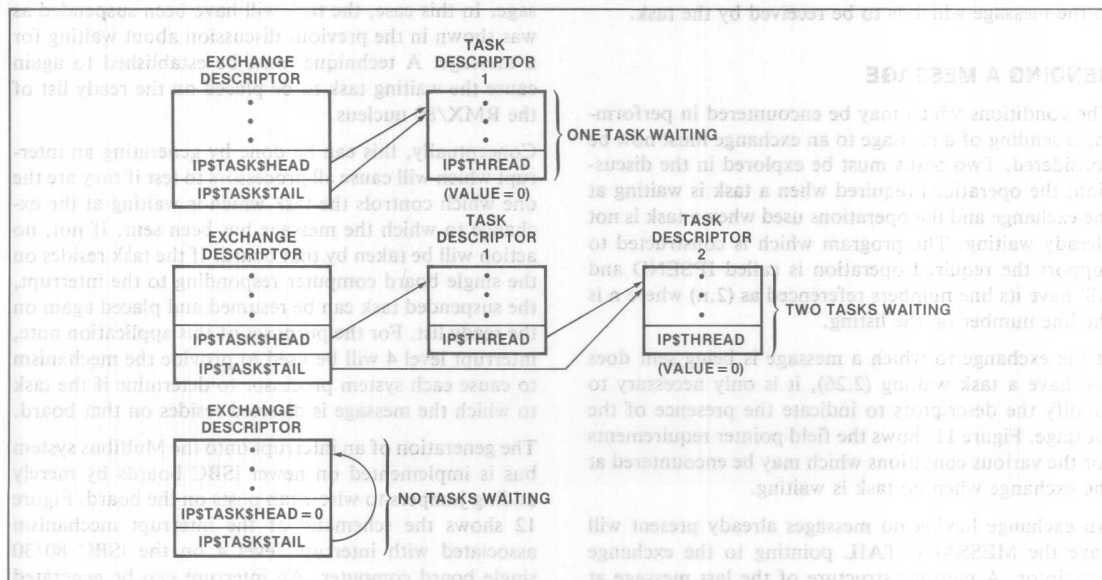


Figure 10. Task Queuing at an Exchange

as shown in Figure 8 and add the new task to the waiting list. The RMX/80 nucleus maintains the address of the task descriptor of the task currently running in a variable, RQACTV. This pointer to the current task must be placed into both the TASK TAIL of the exchange descriptor and into the THREAD of the last task queued onto the exchange (1.29). In addition, the THREAD of the current task should be set to zero to indicate that it will be the last task waiting at the exchange (1.30).

The reader, studying the program listings, might note the condition of the exchange having no tasks waiting when the task is queued up (1.28; 1.29). In this case, the TASK TAIL pointer is clearly pointing to the exchange descriptor and not to a task! The pointer normally placed into the task descriptor's THREAD field will be placed into a field of the exchange descriptor. This potential trouble area is overcome by constructing the fields of both descriptors to have offsets equivalent as shown in Figure 10. The THREAD data will now be placed into the TASK HEAD field as should be done if no tasks are already queued up.

Finally, since the task should no longer compete for system resources until a message is available to it, the program should be placed onto the delay list. However, since this list is maintained by RMX primitive procedures which are unavailable to the user, the same effect can be obtained by constructing a multiprocessor user wait list and then suspending the task (1.32). At this point, the nucleus will activate another task from the ready list and the suspended task will remain in that state until it is resumed. When resumed, program execution will begin at (1.35), where a pointer from the DELAY pointer of the task descriptor will be returned to the calling program. Programs which will activate the task must ascertain that the pointer contains a reference to the message which is to be received by the task.

SENDING A MESSAGE

The conditions which may be encountered in performing a sending of a message to an exchange must now be considered. Two paths must be explored in the discussion; the operations required when a task is waiting at the exchange and the operations used when a task is not already waiting. The program which is constructed to support the required operation is called IPSEND and will have its line numbers referenced as (2.n) where n is the line number on the listing.

If the exchange to which a message is being sent does not have a task waiting (2.26), it is only necessary to modify the descriptors to indicate the presence of the message. Figure 11 shows the field pointer requirements for the various conditions which may be encountered at the exchange when no task is waiting.

An exchange having no messages already present will have the MESSAGE TAIL pointing to the exchange descriptor. A message structure of the last message at the exchange will be defined based upon the MESSAGE

TAIL and the first element of the structure will be set to point to the new message (2.28; 2.29). When no message was already queued, this action will set the MESSAGE HEAD to the new message. When a message was present, the LINK field will be set to point to the new messages. Next, the LINK field of the new message will be set to zero, indicating that it is the last message in the queue (2.30). Operation of the program can then continue.

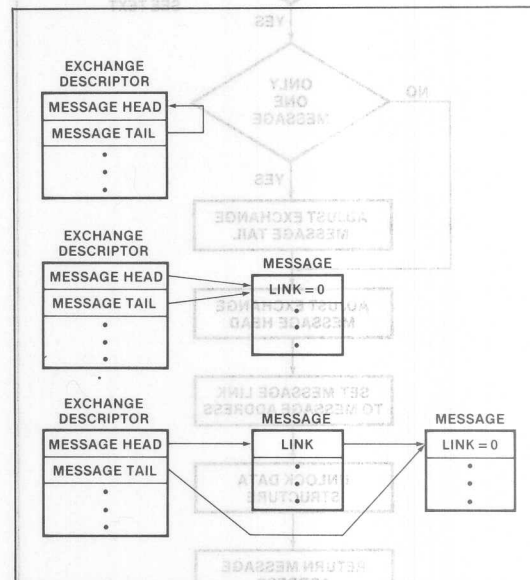


Figure 11. Placing a Message in an Exchange

The operations become more complex when one or more tasks are already waiting at an exchange for a message. In this case, the task will have been suspended as was shown in the previous discussion about waiting for a message. A technique must be established to again cause the waiting task to be placed on the ready list of the RMX/80 nucleus.

Conceptually, this can be done by generating an interrupt which will cause all processors to test if they are the one which controls the task which is waiting at the exchange to which the message has been sent. If not, no action will be taken by that board. If the task resides on the single board computer responding to the interrupt, the suspended task can be resumed and placed again on the ready list. For the purposes of this application note, interrupt level 4 will be used to provide the mechanism to cause each system processor to determine if the task to which the message is directed resides on that board.

The generation of an interrupt onto the Multibus system bus is implemented on newer iSBC boards by merely adding jumpers to wire wrap posts on the board. Figure 12 shows the schematic of the interrupt mechanism associated with interrupt level 4 on the iSBC 80/30 single board computer. An interrupt can be generated onto the Multibus system by toggling bit 7 of the I/O

port. Note that the interrupt is also routed back to the source board, so all processor boards, including the one generating the interrupt, will respond to it.

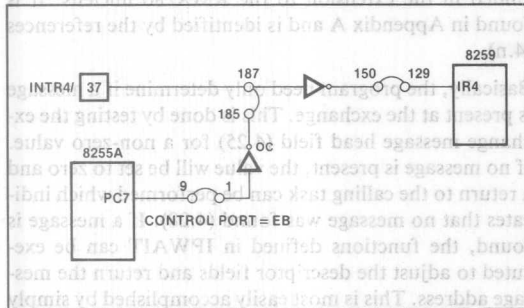


Figure 12. Interrupt Mechanism

A program has been written and is shown in Appendix A which generates the required interrupt onto the system bus. The program is identified by the name SET\$INT and is referenced in this application note with the pointer scheme (3.n) where n is the line number of the code.

Of specific interest in the program listing is the use of software semaphores to provide an indication of the length of time which the interrupt line must be kept active. Semaphore 6 is used to provide mutual exclusion to the interrupt generation mechanism (3.11) and semaphore 5 provides the synchronization mechanism. The

latter semaphore is set prior to establishing an active condition on the interrupt line (3.13). When the processor having the task which is to be activated recognizes the interrupt, it will clear the semaphore level. The interrupt should be held active (low) by the processor board which generates the interrupt until the semaphore has been cleared (3.16; 3.17).

Note that in line (3.14), a processor identification number was stored in the variable PROC\$ID. The technique which can be used to obtain this target processor reference is to store the processor number into the exchange descriptor STATUS field when the task is queued onto the exchange descriptor (1.27). When the message is finally sent to the exchange and the task is taken off the exchange descriptor (2.40), the identification can be stored for use by the interrupted processor. Any processor which receives an interrupt with an identification in PROC\$ID which does not match its own processor identification should ignore the interrupt.

Figure 13 shows the process of dequeuing a task from the exchange descriptor. Since one or more tasks were waiting at the exchange, no messages could exist at the exchange. The first operation (2.36) involves adjusting the pointer to the first task waiting at the exchange to point to the next waiting task (if only one task was waiting, the value of zero will be stored as a pointer). If only one task is waiting (2.36), the exchange task tail is set to point to the exchange (2.37). The message link and the task delay pointers must be set to the address of the

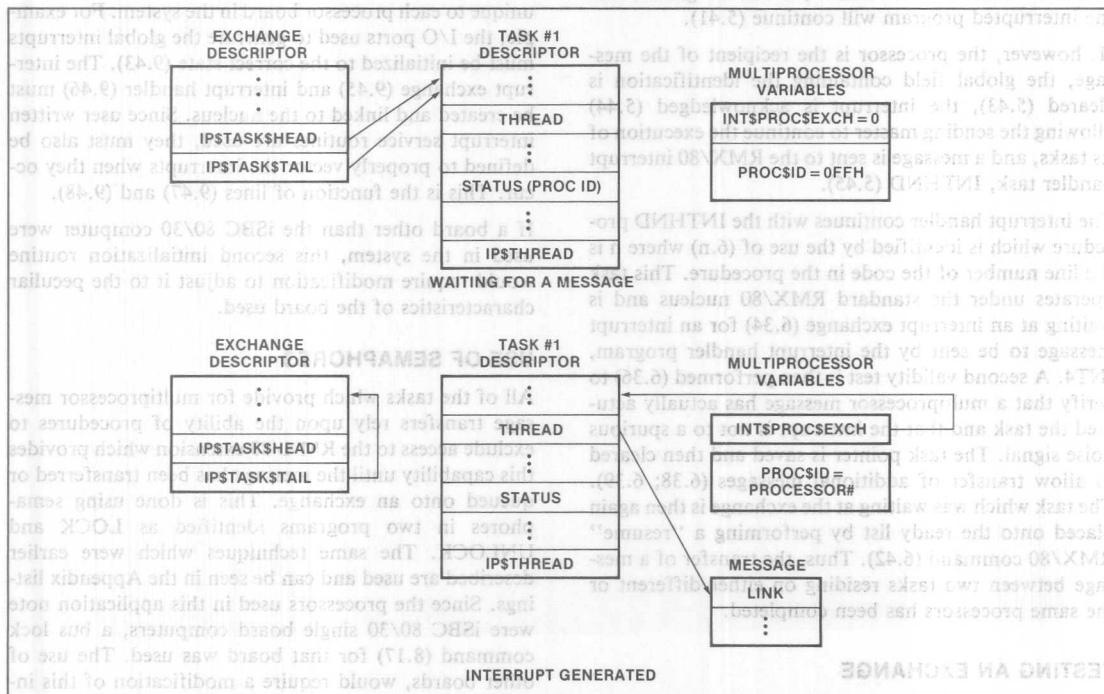


Figure 13. Dequeueing a Task

message (2.38) to support standard RMX/80 protocol. The task delay pointer is used by the receiving task to pass the address of the message when it returns to the active condition (1.33).

Before generating the interrupt which will place the waiting task onto the ready list, parameters which identify the task's processor (2.38) and internal descriptors (2.39) must be placed into a global memory area. As will be seen, this area can then be tested by each processor when an interrupt is received. An interrupt can now be generated which will cause all processors to examine the multiprocessor global memory to determine if the message is directed to a task which resides on that processor board.

RESPONDING TO INTERRUPT REQUEST

As the interrupt is generated, each processor in the system will service the request. An RMX/80 task can be written which waits at the interrupt exchange and which will service the interrupt request. However, in order to optimize the system performance, a user interrupt routine has been provided. A listing of this program, INT4, can be found in Appendix A. Code line numbers for this listing are identified by the nomenclature (5.n) where n is the line number of the code.

The program will first test the target processor identification (5.40) to determine if the task to be placed onto the ready list resides on the board. If the interrupt is found to be for another master, it will be ignored and the interrupted program will continue (5.41).

If, however, the processor is the recipient of the message, the global field containing the identification is cleared (5.43), the interrupt is acknowledged (5.44) allowing the sending master to continue the execution of its tasks, and a message is sent to the RMX/80 interrupt handler task, INTHND (5.45).

The interrupt handler continues with the INTHND procedure which is identified by the use of (6.n) where n is the line number of the code in the procedure. This task operates under the standard RMX/80 nucleus and is waiting at an interrupt exchange (6.34) for an interrupt message to be sent by the interrupt handler program, INT4. A second validity test is then performed (6.36) to verify that a multiprocessor message has actually actuated the task and that the interrupt is not to a spurious noise signal. The task pointer is saved and then cleared to allow transfer of additional messages (6.38; 6.39). The task which was waiting at the exchange is then again placed onto the ready list by performing a "resume" RMX/80 command (6.42). Thus, the transfer of a message between two tasks residing on either different or the same processors has been completed.

TESTING AN EXCHANGE

Many times it is required to test an exchange for the presence of a message without actually waiting if no

message is available at the time. This is the function of the standard RMX/80 procedure RQACPT. A multiprocessor version of this function, IPACPT, was included in the extension to the RMX/80 nucleus. It is found in Appendix A and is identified by the references (4.n).

Basically, the program need only determine if a message is present at the exchange. This is done by testing the exchange message head field (4.25) for a non-zero value. If no message is present, the value will be set to zero and a return to the calling task can be performed which indicates that no message was found (4.28). If a message is found, the functions defined in IPWAIT can be executed to adjust the descriptor fields and return the message address. This is most easily accomplished by simply calling the IPWAIT procedure (4.32).

INITIALIZATION OF DESCRIPTORS

Before the multiprocessor message exchange procedures just described can function properly, various data fields must be initialized just as is done by the RMX/80 nucleus. Two programs have been written to accomplish this task. One, the GLINIT program, initializes the common global variable fields which are common to all processors. This includes such fields as the processor identification (7.14) and the target task pointer (7.9). This program also initializes the extensions which have been made to the exchange descriptors (7.10).

A second program operates on functions which are unique to each processor board in the system. For example, the I/O ports used to generate the global interrupts must be initialized to the correct state (9.43). The interrupt exchange (9.45) and interrupt handler (9.46) must be created and linked to the nucleus. Since user written interrupt service routines are used, they must also be defined to properly vector the interrupts when they occur. This is the function of lines (9.47) and (9.48).

If a board other than the iSBC 80/30 computer were used in the system, this second initialization routine would require modification to adjust it to the peculiar characteristics of the board used.

USE OF SEMAPHORES

All of the tasks which provide for multiprocessor message transfers rely upon the ability of procedures to exclude access to the RMX/80 extension which provides this capability until the message has been transferred or queued onto an exchange. This is done using semaphores in two programs identified as LOCK and UNLOCK. The same techniques which were earlier described are used and can be seen in the Appendix listings. Since the processors used in this application note were iSBC 80/30 single board computers, a bus lock command (8.17) for that board was used. The use of other boards, would require a modification of this instruction to conform to the board design. The bus lock command has no effect on memory which is accessed

using the on-board local bus. In order to provide an effective semaphore, the RAM used for a software semaphore must reside in a memory area which is external to the processor board. For this reason, the dual ported memory of an iSBC 80/30 board cannot be used and memory which is not contained on a board which uses the semaphore must be included in the system.

System Resource Sharing

Many of the advantages of a multiprocessor system can be lost when drivers and peripheral devices are duplicated for each single board computer. For example, consider a system which maintains an on-going inventory system on a mass storage device such as a disk drive. If distributed single board computers are used to support each individual operation of a product as it moves through the process, only a common disk system will allow modification of the data base by each processor. If a means can be found to also share the driver to the disk, considerable programming effort and code space can be saved.

The RMX/80 nucleus is designed to support many special I/O device handlers such as the terminal handler, and the disk file system (DFS). An intermediate level interface can be constructed which will allow any of the system processors to use common drivers on one master board. The program which will be developed in this application note for interfacing with common system drivers is called BIPI. A listing of this program can be found in Appendix B.

Several options exist when selecting a target device which is to contain all the required system resources. One possibility is to arbitrarily pick a processor board and configure the necessary drivers onto it. However, another choice seems to present many more system benefits. This is the dedication of one processor board to the operator interface and then using the iSBC 802 RMX/80 BASIC interpreter on this board. Consider, for a minute, the implications of configuring a system in this manner.

It is a fact that minimal development cost will be incurred if the programming is done in a high level language. Unfortunately, systems involving large amounts of digital input and output, or having to interface with peripheral controllers, do not lend themselves well for programming with a high level language. On the other hand, it is difficult to perform data manipulations of strings and numerical data using programs written in lower level languages. An optimum solution is then to create a system which combines the attributes of both of the language types.

An analogy can be drawn between a conventional single processor system and one which uses multiprocessors. In the system which uses a single processor, assembly language routines can be written for minimum code size or time critical functions and can be linked with PL/M or FORTRAN programs to produce the final object

module. A multiprocessor system can be thought of in the same manner. Here, individual functions are represented by a complete single board computer, each of which can use some combination of the available languages. System design can emphasize the positive attributes of each language which is available.

The iSBC 802 BASIC package can be included in a multiprocessor system much the same as a common subroutine in smaller systems. Various other processor boards can use the I/O capabilities of the BASIC interpreter as needed. In particular, the DFS (Disk File System) and terminal handler can be shared as needed by each board.

Because the I/O drivers contained in the BASIC Interpreter use standard RMX/80 exchanges, they cannot be directly accessed using the multiprocessor interface which has been developed earlier in this application note. A special intermediate interface is required which will interface with the iSBC 802 software. This is the function of the program, BIPI, which will be linked together with the other tasks contained on the single board computer used for the BASIC interpreter. Certainly, the concepts which will be used to develop the interface to the common I/O services of the iSBC 802 software can easily be extended to support other packages such as the iSBC 801 FORTRAN run-time package.

Requests to have a service performed by the BIPI module will be received in the form of a message to the global exchange, BIPI\$EX. A definition of services which may be required by a remote processor will provide the basis for a definition of message types which will be supported by the interface.

A fundamental request will be to gain access to the operator keyboard or display terminal. Thus two message types can be defined to read from the terminal keyboard (READ\$TYPE) and to output to the display (WRITE\$TYPE). To provide process tasks with the ability to examine and modify data which is stored on the disk drive media, a set of commands must also be generated which interface to DFS. Five command types will prove to be sufficient for this task. Corresponding to the RMX DFS disk message types, the program will define disk open operations (DFS\$OPN) and close operations (DFS\$CLS). Positioning capabilities are supported by a seek command (DFS\$SK). Finally, read and write requests are supported by the read (DFS\$RD) and write (DFS\$WT) commands.

Examination of the program listing will indicate that disk request message types use the type numbers from 72 to 79. This provides a simple method of distinguishing a disk request from a terminal operation message. The numbers were chosen such that subtraction of 64 from the message type will yield the message type which must actually be sent to the DFS exchanges by BIPI.

In many cases, it may be desirable to suspend the BASIC program (or the FORTRAN program) while

peripheral I/O resources are being shared by another processor. An example might be when a task is reporting an alarm condition to the operator and requires some interactive communications with him. Certainly, these communications cannot be intermixed with messages from the program which was running on the terminal. Therefore, two additional commands have been provided which will suspend (SUSBAS) and resume (RESBAS) the task which might have been competing for a system resource.

Study of the program listing for BIPI will provide the reader with an insight into the use of the multiprocessing RMX/80 extensions which have been developed in this application note. Both standard RQ . . . calls and multiprocessor IP . . . calls are integrated into the task. When the concepts are clear, a real application example can be examined to see how multiprocessing can assist in the solution of an otherwise very complex design problem.

IV. APPLICATION EXAMPLE

The previous development of multiprocessing extensions to the RMX/80 nucleus allows the application example development to continue. A solution using multiprocessor techniques can be easily implemented using these concepts and the extensive line of Intel single board computer products. The results can be examined to verify that multiprocessing has provided a better solution than would have been gained if only one processor had been used.

The merits and deficiencies of multiprocessing must be examined to ascertain that the application is likely to benefit from a solution which uses more than one processor. Certainly, many functions can be better performed using a higher speed microcomputer on a single

board. The decision to use multiprocessing must be made on a cost/performance basis.

The application defined at the beginning of this application note provides an excellent example of the multi-master solution. Each functional section of the application is developed into an actual board implementation which includes all resources required for its support.

Supervisor Implementation

The supervisory functions are likely to require extensive computational functions, report generation and operator interaction. It is also likely that, as the management gains experience with the system, the functions required and the algorithms used will be constantly modified. These operations are not generally real-time driven so extremely high system throughputs will not be required. The Intel iSBC 802 BASIC package running on a single board computer will provide a solution to these requirements. In terms of hardware design, the supervisory functions can be considered without regard to any control functions. The primary interface can be through data bases stored on a mass storage device and with the "peek" and "poke" instructions where necessary. The system configuration for the supervisory function can be seen in Figure 14.

Because the supervisor will rely extensively upon the interaction with the operator and the mass storage devices, it is natural that it have associated with it the terminal handler and the DFS system. The addition of the BIPI interface task to the BASIC interpreter package will allow the control tasks to easily interface to the inventory and control data bases as required. Figure 15 shows the primary tasks which operate on the supervisor control processor. Note that there exists two operating tasks, BASIC and BIPI, along with support functions of the DFS and terminal handler. Any program which is

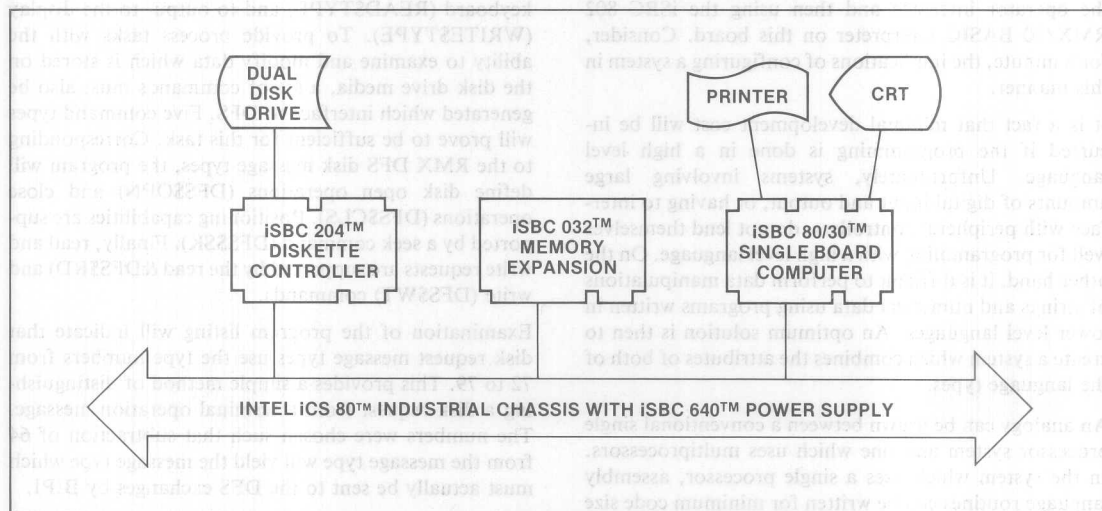


Figure 14. Supervisory Configuration

being run under BASIC will essentially become an extension of the existing interpreter task.

Programs can be written which will provide the required algorithms to implement the inventory control system and production scheduling of the system using a high level language which may easily be modified as required by the future system requirements.

The rest of the system consists of control elements. They may easily be thought of as supervisor subroutines which are called as required and which have parameters passed with the call. Unlike conventional designs, these subroutines will reside on different processor boards.

Weighing Ingredients

One functional task has been defined as weighing the ingredients according to a formula in order to provide the correct ratio of ingredients to produce the final product. As with most functional areas, a closer examination shows that this involves rather complex relationships and breaks down into many smaller tasks. Figure 16 shows the logical flow of operations which are required to weigh the ingredients into their proper ratios. From the information contained in this flow diagram, tasks and exchanges necessary to implement the function can be defined and the coding generated.

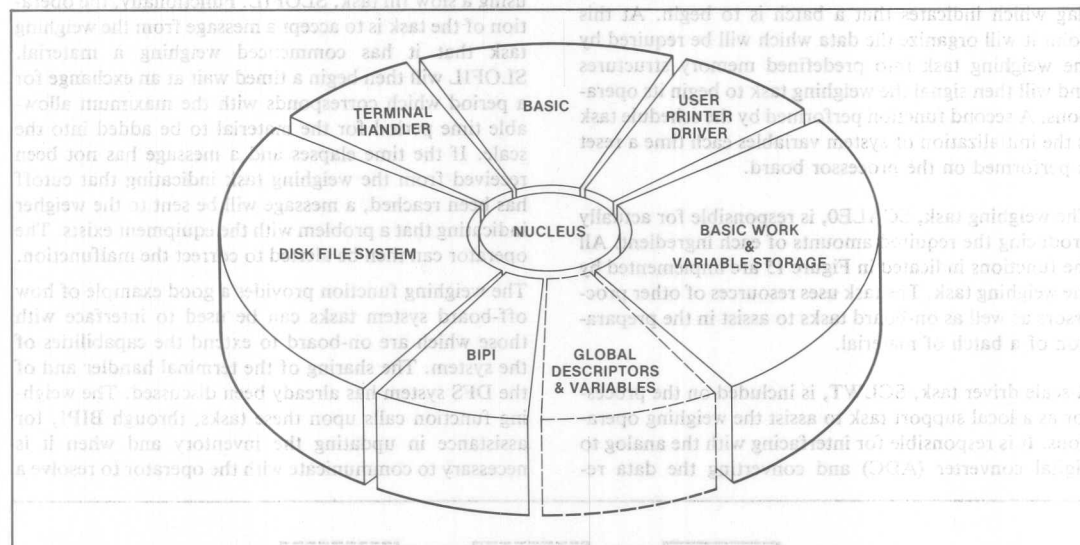


Figure 15. Supervisor Software Structure

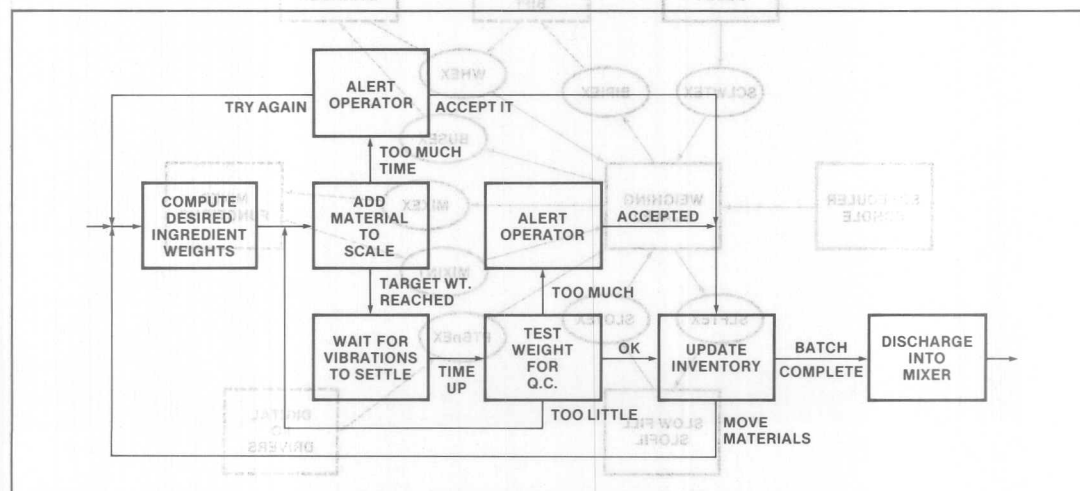


Figure 16. Weighing Logic Flow

The tasks which were defined for this application example are shown in Figure 17. The task communication paths through exchanges has also been illustrated in the figure. Basically, the weighing function consists of four on-board tasks and calls upon tasks contained on other boards as is required using the global exchanges created according to the structure defined earlier in this application note. Some time should be taken to explain the functions of each task and how they relate with tasks of other processors in the multiprocessor design solution.

The schedule task, SCHDLE, is the interface between the operator commands and the weighing functions. This task will wait for a POKE command to pass it a flag which indicates that a batch is to begin. At this point it will organize the data which will be required by the weighing task into predefined memory structures and will then signal the weighing task to begin its operations. A second function performed by the schedule task is the initialization of system variables each time a reset is performed on the processor board.

The weighing task, SCALE0, is responsible for actually producing the required amounts of each ingredient. All the functions indicated in Figure 15 are implemented by the weighing task. The task uses resources of other processors as well as on-board tasks to assist in the preparation of a batch of material.

A scale driver task, SCLWT, is included on the processor as a local support task to assist the weighing operations. It is responsible for interfacing with the analog to digital converter (ADC) and converting the data re-

ceived from it into engineering units which represent the actual weight of material which is in the scale. The scale weights are passed between two tasks using an exchange (SCLWTEX) to provide mutual exclusion.

In weighing applications, many times a feeder will malfunction and not deliver material into a scale when directed to do so. If no provision is incorporated into a system design to detect this condition, an infinite time period could be required to weigh a material and, needless to say, the system throughput would be severely degraded. A slow fill detection algorithm is normally incorporated into weighing systems to provide this service. For this application, this function is generated using a slow fill task, SLOFIL. Functionally, the operation of the task is to accept a message from the weighing task that it has commenced weighing a material. SLOFIL will then begin a timed wait at an exchange for a period which corresponds with the maximum allowable time period for the material to be added into the scale. If the time elapses and a message has not been received from the weighing task indicating that cutoff has been reached, a message will be sent to the weigher indicating that a problem with the equipment exists. The operator can then be alerted to correct the malfunction.

The weighing function provides a good example of how off-board system tasks can be used to interface with those which are on-board to extend the capabilities of the system. The sharing of the terminal handler and of the DFS system has already been discussed. The weighing function calls upon these tasks, through BIPI, for assistance in updating the inventory and when it is necessary to communicate with the operator to resolve a

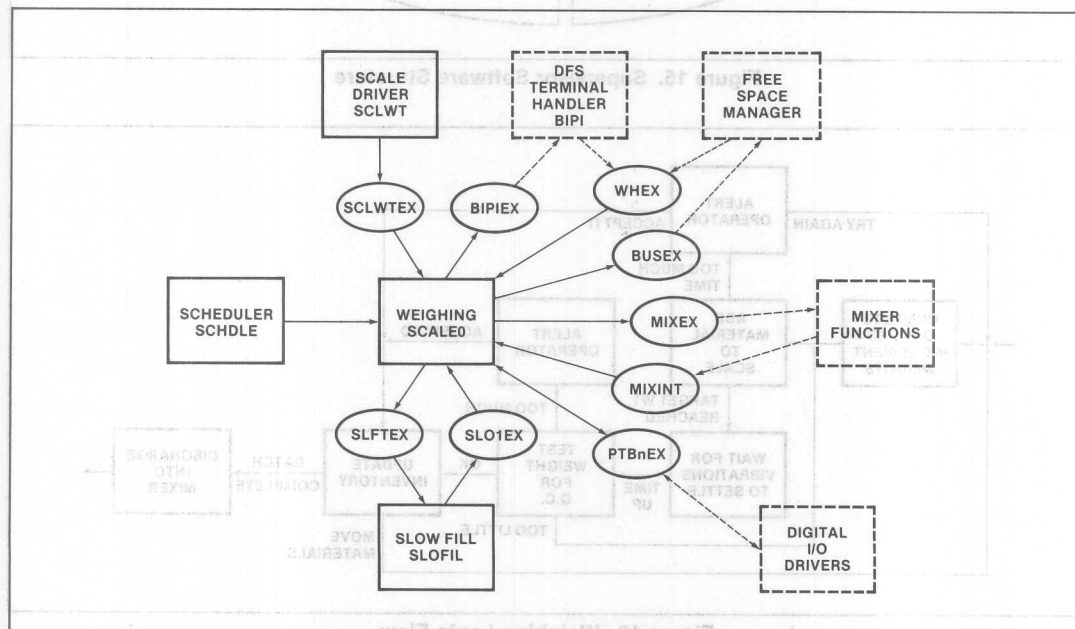


Figure 17. Weighing Task/Exchange Relationships

system problem. Other examples can also be seen. For example, the decision was made to use a common digital I/O board for all process functions. This gives the ability for access to the control and sensor devices to both the controlling tasks and to the operator for performing maintenance programs or even operating the system in a semi-automatic mode should a controller fail. All I/O data is moved through exchanges, PTBnEX, and the actual I/O is performed by a task which resides on another board.

Another example of multiprocessor communications involves message passing between the weighing tasks and the mixing tasks contained on another board. The weighing task is responsible for discharging the material into the mixer, but it certainly cannot do so unless it ascertains that the mixer is empty and available. It does this by testing the exchange, MIXINT, for the presence of a message indicating that the mixer is available. The mixer tasks, as will be seen, support this exchange by removing the message when no material should be added into the mixer and placing a message into the exchange when a mixer is ready. Further communications with the mixer are required because the mixer task must be synchronized with the discharging of material into it by the weighing function. This is provided by message transfer through global exchange, MIXEX.

Finally, because only a limited amount of RAM is available in a particular system, the free space manager was used in this application to allocate blocks of global memory to individual processor boards. This allocation is handled by sending messages to the global exchange, BUSEX, which will be received by an intermediate task on another processor and will result in a block of memory being allocated to the weighing task for message generation.

Other Application Tasks

A processor was dedicated to support each of the remaining three application processes. It would be repetitious to again detail each subtask which makes up each functional area. The same techniques which were demonstrated in defining the weighing application can be used to define the generalized functions and to break these functions into codable tasks.

Data is transferred between processors in the form of messages which occupy RAM area. The use of the RMX/80 Free Space Manager provides a global tool to allocate and reclaim this memory area. The Free Space Manager was implemented on the mixing processor because substantial amounts of ROM remained on-board after coding all the required tasks. This ability to select from many processors the location for global resources is an important tool in multiprocessor applications.

Total Application Configuration

Figure 18 shows the functional software implementation for the entire chemical application chosen for this appli-

cation note. The approach has demonstrated that a relatively complex application can easily and quickly have a control solution generated using multiprocessing concepts. The ability to extend the multitasking capabilities of the RMX/80 nucleus led toward the creation of modular software.

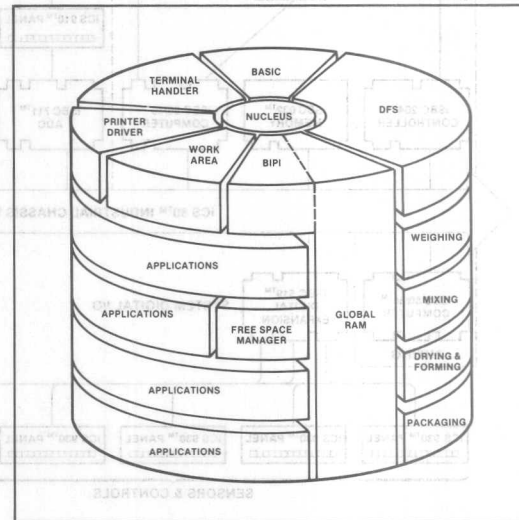


Figure 18. Software Configuration

In addition to allowing the generation of modular software, the approach taken on this example has allowed a modular approach to the hardware implementation. Figure 19 provides the complete hardware block diagram of the final implemented system. Each functional section was designed as though it were the only required element to create a solution to the control problem. Indeed, even after start-up, a functional module can be removed from the system without affecting the operation of remaining modules. The concept can be extended to include the capability of adding new processes to the total system without having to disrupt the existing production flow control.

V. CONCLUSION

This application note has demonstrated how a user can extend the concepts of multitasking into a multiprocessing/multitasking environment. The built-in multiprocessing capabilities of the various Intel single board computers have been explained and their implementations demonstrated through examples.

Resource sharing between processors has been discussed. In particular, the use of a high level language such as BASIC or FORTRAN as a supervisor has been explained and shown in an application. Since these software packages contain the resources required for disk support and terminal I/O, a program has been developed which will allow other processors to share the RMX/80 drivers.

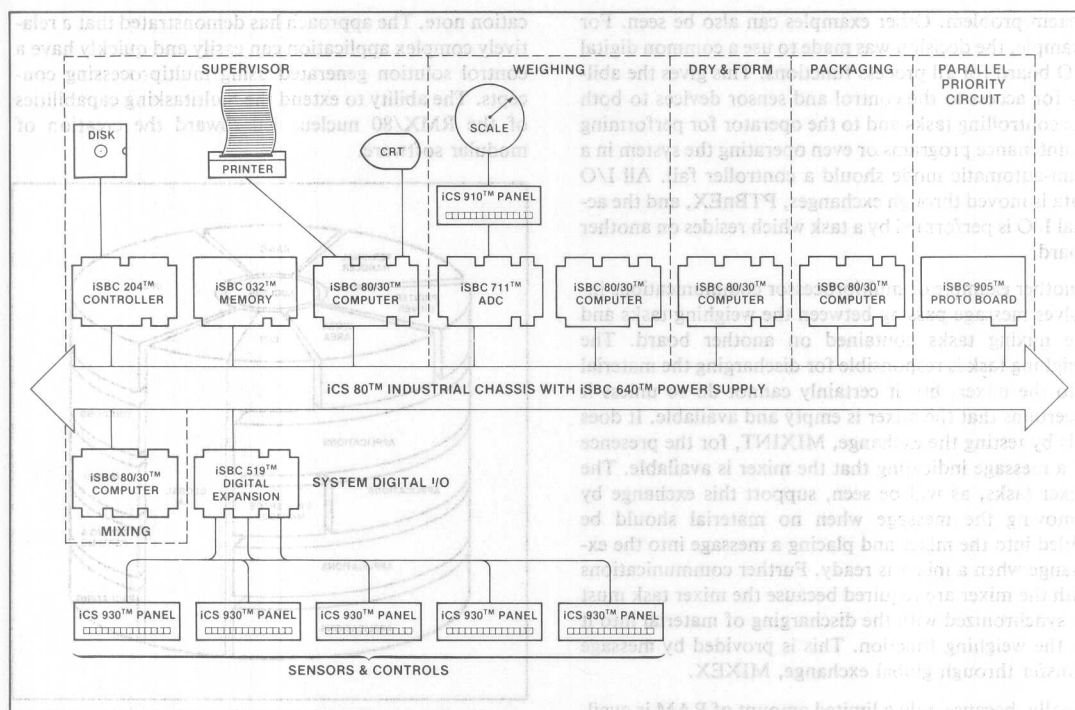


Figure 19. Hardware Block Diagram

Finally, an application has been selected, and a control solution developed using the concepts derived in the application note. The operation of the RMX/80 extensions have been tested and their operation verified.

Multimaster operations are possible because of many built-in features of the Intel products and the extensions to the RMX/80 nucleus used in this application. A summary of these features and the corresponding product is:

Multitasking Capability —

Furnished by the RMX/80 nucleus. It allows the user to share the on-board resources of a processor board between multiple tasks or functions.

Message Transfer Capabilities —

Also furnished by the RMX/80 nucleus. This function allows data to be passed between tasks to provide both an information exchange and a means of task synchronization.

Free Space Management —

The RMX/80 executive provides the ability to share memory between various tasks. Memory which is not constantly required can be used by more than one user, then returned to the memory pool.

Terminal Handler —

The RMX/80 terminal handler provides a convenient resource for supporting the communications with the operator via a CRT terminal.

Disk File System —

The RMX/80 Disk File System (DFS) provides a common resource which allows tasks to access or to store data on a diskette.

Interprocessor Communication Software —

The software developed in this application note provides an extension to the RMX/80 nucleus which allows tasks to reside on any multimaster board connected to the Multibus system bus.

Bus Arbitration Logic —

The Intel single board computers used in this application incorporate bus arbitration logic which controls the bus access to the Multibus system bus.

Parallel Priority Logic —

The application uses a parallel priority logic network which is constructed on a prototype board. In conjunction with the bus arbitration logic on each multimaster board, it controls the requests for data transfers over the Multibus system bus.

Bus Lock —

The capability of assuming exclusive control of the Multibus system bus by using the bus lock mechanism allows the implementation of software semaphores. These semaphores are used to provide mutual exclusion of certain physical resources.

BASIC Interpreter —

The use of the iSBC 802 BASIC interpreter greatly reduces the amount of time required for the generation of report programs and of interactive communications with the operator.

PL/M 80 High Level Language —

This software language provides an efficient means of coding the application software where intensive bit manipulations are required. The structured language also provides a rapid development cycle.

The reader should have considerable insight into possible system level solutions to applications which he may face in his particular expertise. The increased productivity which results from applying the engineering/programming resources into application oriented efforts can easily be seen. The development costs which can be saved by using modular software and high integration level board products can certainly justify the use of these system solutions.

I would like to extend my gratitude to Steve Verleye for his valuable assistance in generating the multiprocessor communications programs used in this application note.

APPENDIX A
MULTIPROCESSOR MESSAGE TRANSFER PROGRAMS

The reader should have considerable insight into possible system level solutions to applications which he may face in his particular expertise. The increased productivity which results from applying the engineering programming resources into application oriented efforts can easily be seen. The development costs which can be saved by using modular software and high integration level board products can certainly justify the use of these system solutions.

I would like to extend my gratitude to Steve Verheye for his valuable assistance in generating the multiprocessor communications programs used in this application note.

The use of the BASIC 802 BASIC interpreter greatly reduces the amount of time required for the generation of report programs and of interactive communications with the operator.

PLM 80 High Level Language —

The software language provides an efficient means of coding the application software where intensive bit manipulations are required. The structured language also provides a rapid development cycle.

APPENDIX A MULTIPROCESSOR MESSAGE TRANSFER PROGRAMS

PUBLIC PROCEDURE: IPWAIT. LAST CHANGED 11/07/79

This procedure provides the inter-processor wait facility. If a message is available at the given exchange it is taken off the exchange and its address is returned to the calling routine. If no message is available, the processor ID is encoded in the status field, the task address is queued on the exchange list, and the task is suspended. When a message is sent to the exchange by another processor, the processor ID is used to send the task and message addresses into the incoming task queue for the processor and then interrupt it. The service routine thus activated sets the delay field of the indicated task equal to the message address and RESUMES the task. The task then simply RETURNS the stored message address.

```

*/
$include(:fl:global.ext)
/* Declaration of global inter-processor data
   structures */
2 1 = declare
    = proc$pid byte external,
    = my$pid byte external,
    = in$que address external;
3 1 = declare
    = my$proc$pid literally 'my$pid',
    = int$proc$exch literally in$que';
$include(:fl:sema.ext)
4 1 = lock: procedure (sema$pid) external;
5 2 = declare sema$pid byte;
6 2 = end;
7 1 = unlock: procedure (sema$pid) external;
8 2 = declare sema$pid byte;
9 2 = end;
$include(suspnd.ext)
10 1 = RQSUSP:
    = PROCEDURE (T$PTR) EXTERNAL;
11 2 = DECLARE T$PTR ADDRESS;
    =
12 2 = END RQSUSP;
$include(msg.elt)
13 1 = DECLARE MSG$HDR LITERALLY '
    = LINK ADDRESS,
    = LENGTH ADDRESS,
    = TYPE BYTE,
    = HOME$EXCHANGE ADDRESS,
    = RESPONSE$EXCHANGE ADDRESS';
14 1 = DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE (
    = MSG$HDR,
    = REMAINDER(1) BYTE)';

```

```

$include(:fl:iptask.elt)
15 1 = DECLARE IP$TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
    = DELAY$LINK$FORWARD ADDRESS,
    = DELAY$LINK$BACK ADDRESS,
    = THREAD ADDRESS,
    = DELAY ADDRESS,
    = EXCHANGE$ADDRESS ADDRESS,
    = SP ADDRESS,
    = MARKER ADDRESS,
    = PRIORITY BYTE,
    = STATUS BYTE,
    = NAME$PTR ADDRESS,
    = TASK$LINK ADDRESS,
    = IP$THREAD ADDRESS);
$include(:fl:ipexch.elt)
16 1 = DECLARE IP$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
    = MSG$HEAD ADDRESS,
    = MSG$TAIL ADDRESS,
    = TASK$HEAD ADDRESS,
    = TASK$TAIL ADDRESS,
    = NEXT$EXCH ADDRESS,
    = RESERVED (10) BYTE,
    = IP$TASK$HEAD ADDRESS,
    = IP$TASK$TAIL ADDRESS);
$include(common.elt)
17 1 = DECLARE TRUE LITERALLY '0FFH';
18 1 = DECLARE FALSE LITERALLY '00H';
19 1 = DECLARE BOOLEAN LITERALLY 'BYTE';
20 1 = DECLARE FOREVER LITERALLY 'WHILE 1';
    $list
21 1 declare
    RQACTV address external;
22 1 IPWAIT: procedure (exch$adr, delay) address reentrant public;
23 2 declare
    (exch$adr, delay, temp$msg$ptr, last$task$ptr) address,
    (temp$msg based temp$msg$ptr) msg$descriptor,
    (last$task based last$task$ptr) ip$task$descriptor,
    (exch based exch$adr) ip$exchange$descriptor,
    (task based RQACTV) ip$task$descriptor;

    /* lock exchange data structure */
24 2 call lock(7);

25 2 if (temp$msg$ptr = exch.msg$head) = 0 then
26 2 do;
    /* no message at exchange, queue task up */
27 3 task.status = task.status OR shl(my$proc$id, 3);
28 3 last$task$ptr = exch.ip$task$tail;
29 3 last$task.ip$thread, exch.ip$task$tail = RQACTV;
30 3 task.ip$thread = 0;
31 3 call unlock(7);
32 3 call RQSUSP(RQACTV);
33 3 return task.delay;
34 3 end;

35 2 else do;

```

```

/* There is a message here. Dequeue it and return
its address */
36 3      if(exch.msg$head:=temp$msg$link) = 0 then
37 3      exch.msg$tail:=exch;
38 3      temp$msg.link:=temp$msg$ptr;
39 3      call unlock(7);
40 3      return temp$msg$ptr;
41 3      end;

42 2      end; /* of procedure */

43 1      end IPWAIT;

MODULE INFORMATION:
CODE AREA SIZE      = 00F6H 246D
VARIABLE AREA SIZE = 0000H 0D
MAXIMUM STACK SIZE = 000AH 10D
133 LINES READ
0 PROGRAM ERROR(S)

1      $title('IPSEND routine, version 1.3')
      IPSEND:
      do;
      /*
      ++++++

      PUBLIC PROCEDURE: IPSEND. LAST CHANGED 02/11/80

      This routine provides the interprocessor send facility.
      If no tasks are waiting for the message, the message is
      queued on the exchange. If a task is waiting, it is taken
      off the queue, given the address of the message, and
      queued on the interprocessor exchange.
      Following this an interrupt is generated to
      signal the responsible processor that a task is ready.

      ++++++
      */
      $include(susnd.ext)
2 1 =  RQSUSP:
      =  PROCEDURE (T$PTR) EXTERNAL;
3 2 =  DECLARE T$PTR ADDRESS;
      =
4 2 =  END RQSUSP;
      $include(msg.elt)
5 1 =  DECLARE MSG$HDR LITERALLY
      =  LINK ADDRESS,
      =  LENGTH ADDRESS,
      =  TYPE BYTE,
      =  HOME$EXCHANGE ADDRESS,
      =  RESPONSE$EXCHANGE ADDRESS';
6 1 =  DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
      =  MSG$HDR,

```

```

7 1 = $include(:f1:iptask.elc)
    = DECLARE IP$TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
    = DELAY$LINK$FORWARD ADDRESS,
    = DELAY$LINK$BACK ADDRESS,
    = THREAD ADDRESS,
    = DELAY ADDRESS,
    = EXCHANGE$ADDRESS ADDRESS,
    = SP ADDRESS,
    = MARKER ADDRESS,
    = PRIORITY BYTE,
    = STATUS BYTE,
    = NAME$PTR ADDRESS,
    = TASK$LINK ADDRESS,
    = IP$THREAD ADDRESS)';
    include(:f1:ipexch.elc)
8 1 = DECLARE IP$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
    = MSG$HEAD ADDRESS,
    = MSG$TAIL ADDRESS,
    = TASK$HEAD ADDRESS,
    = TASK$TAIL ADDRESS,
    = NEXT$EXCH ADDRESS,
    = RESERVED (10) BYTE,
    = IP$TASK$HEAD ADDRESS,
    = IP$TASK$TAIL ADDRESS)';
    $include(common.elc)
9 1 = DECLARE TRUE LITERALLY '0FFH';
10 1 = DECLARE FALSE LITERALLY '00H';
11 1 = DECLARE BOOLEAN LITERALLY 'BYTE';
12 1 = DECLARE FOREVER LITERALLY 'WHILE 1';
    $list
    $include(:f1:sema.ext)
13 1 = lock: procedure (sema$id) external;
14 2 = declare sema$id byte;
15 2 = end;
16 1 = unlock: procedure (sema$id) external;
17 2 = declare sema$id byte;
18 2 = end;
    $include(:f1:global.ext)
    = /* Declaration of global inter-processor
      data structures */
19 1 = declare
    = proc$id byte external,
    = my$pid byte external,
    = in$que address external;
20 1 = declare
    = my$proc$id literally 'my$pid',
    = int$proc$exch literally 'in$que';
21 1 = set$int: procedure external;
22 2 = end set$int;
23 1 = IPSEND: procedure (exch$ptr,msg$ptr) reentrant public;
24 2 = declare
    (exch$ptr,msg$ptr,last$msg$ptr,task$ptr) address,

```

```

*****
      (exch based exch$ptr) ip$exchange$descriptor,
      (msg based msg$ptr) msg$descriptor,
      (last$msg based last$msg$ptr) msg$descriptor,
      (task based task$ptr) ip$task$descriptor;
*****
      /* get access to data structures */ \

25      2      call lock(7);
      /* check to see if any tasks are waiting */

26      2      if (task$ptr:= exch.ip$task$head)=0 then
27      2      do; /* no tasks waiting; queue message */
28      3      last$msg$ptr:=exch.msg$tail;
29      3      last$msg.link.exch.msg$tail=msg$ptr;
30      3      msg.link=0;
31      3      call unlock(7);
32      3      return;
33      3      end;

34      2      else do;
      /* unqueue task and send it to proper processor */
35      3      if(exch.ip$task$head:= task.ip$thread)= 0 then
36      3      exch.ip$task$tail:=exch$ptr;
37      3      msg.link,task.delay=msg$ptr;

      /* queue task on the interprocessor exchange */

38      3      proc$ptr:=shr(task.status.3) and 07H;
39      3      int$proc$exch:=task$ptr;
40      3      task.ip$thread:=0;
41      3      call set$int;
42      3      call unlock(7);
43      3      return;
44      3      end;
45      2      end; /* of procedure */
46      1      end IPSEND;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0108H      264D
VARIABLE AREA SIZE  = 0000H      0D
MAXIMUM STACK SIZE  = 0000H      12D
130 LINES READ
0 PROGRAM ERROR(S)

```

```
$title('set interrupt routine')
```

```

1      Set$int:
      do;
      /*

```

```
*****
```

```
PUBLIC PROCEDURE: SET$INT. LAST CHANGED 2/11/80.
```

```

Set interrupt routine. Generates a level
four interrupt on the bus. The semaphore associated

```



```

with the proc$pid field is left set until the interrupt
is serviced. An semaphore, level 5, is set and used as
an indicator that the interrupt has been acknowledged.
+++++
/* get access to data structures */
$include(:fl:global.ext)
= /* Declaration of global inter-processor
/* check to see if data structures */
=
2 1 declare
= proc$pid byte external;
= my$pid byte external;
= in$queue address external;
3 1 declare
= my$proc$pid literally 'my$pid',
= int$proc$exch literally 'in$queue';
$include(:fl:sema.ext)
4 1 = lock: procedure (sema$pid) external;
5 2 = declare sema$pid byte;
6 2 = end;
/* undqueue task and send it to proper processor
if(exch.lock$head = task.lock$head) then
7 1 = unlock: procedure (sema$pid) external;
8 2 = declare sema$pid byte;
9 2 = end;
/* queue task on the interprocessor exchange
10 1 set$int: procedure public;
proc$pid=shr(task.status.3) and 07H;
in$proc$exch=task$pid;
11 2 call lock(6);
12 2 call lock(5);
13 2 output(0EBH)=0FH;
14 2 call lock(5);
15 2 output(0EBH)=0EH;
16 2 call unlock(5);
17 2 call unlock(6);
18 2 return;
19 2 end;
20 1 end set$int;
MODULE INFORMATION:
CODE AREA SIZE = 0108H
VARIABLE AREA SIZE = 0000H
MAXIMUM STACK SIZE = 0002H
170 LINES READ
0 PROGRAM ERROR(S)
49 LINES READ
0 PROGRAM ERROR(S)

```

```

$title('IPACPT routine')
1 IPACPT:
do;
/*
+++++
PUBLIC PROCEDURE: IPACPT. LAST CHANGED 10/12/79
This routine implements the interprocessor accept facility.

```

If no message is queued at the exchange, a zero is returned.
Else, a call is made to IPWAIT to retrieve the first message
on the queue.

```

*****
*/
$include(suspcnd.ext)
2 1 = RQSUSP:
    = PROCEDURE (T$PTR) EXTERNAL;
3 2 = DECLARE T$PTR ADDRESS;
    =
4 2 = END RQSUSP;
$include(msg.elt)
5 1 = DECLARE MSG$HDR LITERALLY
    = LINK ADDRESS,
    = LENGTH ADDRESS,
    = TYPE BYTE,
    = HOME$EXCHANGE ADDRESS,
    = RESPONSE$EXCHANGE ADDRESS';
6 1 = DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE (
    = MSG$HDR,
    = REMAINDER(1) BYTE)';
$include(task.elt)
7 1 = DECLARE TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
    = DELAY$LINK$FORWARD ADDRESS,
    = DELAY$LINK$BACK ADDRESS,
    = THREAD ADDRESS,
    = DELAY ADDRESS,
    = EXCHANGE$ADDRESS ADDRESS,
    = SP ADDRESS,
    = MARKER ADDRESS,
    = PRIORITY BYTE,
    = STATUS BYTE,
    = NAME$PTR ADDRESS,
    = TASK$LINK ADDRESS)';
$include(exch.elt)
8 1 = DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
    = MSG$HEAD ADDRESS,
    = MSG$TAIL ADDRESS,
    = TASK$HEAD ADDRESS,
    = TASK$TAIL ADDRESS,
    = NEXT$EXCH ADDRESS)';
$include(common.elt)
9 1 = DECLARE TRUE LITERALLY '0FFH';
10 1 = DECLARE FALSE LITERALLY '00H';
11 1 = DECLARE BOOLEAN LITERALLY 'BYTE';
12 1 = DECLARE FOREVER LITERALLY 'WHILE 1';
$include(:fl:sema.ext)
13 1 = lock: procedure (sema$ld) external;
14 2 = declare sema$ld byte;
15 2 = end;
16 1 = unlock: procedure (sema$ld) external;
17 2 = declare sema$ld byte;
18 2 = end;
$list

```

```

19 1  IPWAIT: procedure (exch$adr, delay) address external;
20 2  declare (exch$adr, delay) address;
21 2  end IPWAIT;

22 1  IPACPT: procedure (exch$ptr) address reentrant public;

23 2  declare
      exch$ptr address,
      (exch based exch$ptr) exchange$descriptor;

24 2  call lock(7);
25 2  if exch.msg$head = 0 then
26 2  do;
27 3      call unlock(7);
28 3      return 0;
29 3  end;
30 2  else do;
31 3      call unlock(7);
32 3      return IPWAIT(exch$ptr, 0);
33 3  end;
34 2  end; /* of procedure */
35 1  end IPACPT;

MODULE INFORMATION:
CODE AREA SIZE      = 003BH      59D
VARIABLE AREA SIZE = 0000H      0D
MAXIMUM STACK SIZE = 0004H      4D
90 LINES READ
0 PROGRAM ERROR(S)

      $title('Level 4 interrupt handler')
      $nointvector
1      Int$proc:
      do;

      /*
      ++++++
      PUBLIC PROCEDURE: SET$VEC$4. LAST CHANGED 2/11/80

      This routine fields all level 4 interrupts. If the global
      proc$id matches the local processor ID RQISND is called
      to awaken the int$hnd task and the interrupt semaphore is
      cleared. If the processor ID does not match, the interrupt
      is ignored.

      ++++++
      */

      $nolist

38 1  declare
      rql4ex (15) byte external;

39 1  set$vec$4: procedure interrupt 4 public;

40 2  if proc$id <> my$proc$id then
41 2  call rqendi;

```

```

42 2      else do;
43 3          proc$Id=0FFH;
44 3          call unlock(5);
45 3          call rqisnd(.rq14ex);
46 3      end;
47 2      return;
48 2      end; /* of int$proc */
49 1      end int$proc;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 002AH      42D
VARIABLE AREA SIZE  = 0000H      0D
MAXIMUM STACK SIZE  = 000AH      10D
110 LINES READ
0 PROGRAM ERROR(S)

```

```

1          $title('Interrupt Handler')
          Int$handler:
              do;
                  /*
+++++
PUBLIC PROCEDURE: INT$HANDLER. LAST CHANGED 2/11/80

Interprocessor interrupt handler task. When notified by the
Set$vec$4 routine that the In$queue contains a task-message
pair for this processor, the Int$handler task unqueues
the task descriptor and resumes the task where it was
suspended.

+++++
*/
          $include(:fl:global.ext)
          = /* Declaration of global inter-processor
            data structures */
2 1 = declare
          = proc$Id byte external,
          = my$pid byte external,
          = in$que address external;
3 1 = declare
          = my$proc$Id literally 'my$pid',
          = int$proc$exch literally 'in$que';
          $include(:fl:sema.ext)
4 1 = lock: procedure (sema$Id) external;
5 2 = declare sema$Id byte;
6 2 = end;
          =
7 1 = unlock: procedure (sema$Id) external;
8 2 = declare sema$Id byte;
9 2 = end;
          $include(synch.ext)
10 1 = RQSEND:
          = PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
11 2 = DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
          =

```

```

12 2 = END RQSEND;
13 1 = RQWAIT:
    = PROCEDURE (EXCHANGE$POINTER, DELAY) ADDRESS EXTERNAL;
14 2 = DECLARE (EXCHANGE$POINTER, DELAY) ADDRESS;
15 2 = END RQWAIT;
16 1 = RQACPT:
    = PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
17 2 = DECLARE EXCHANGE$POINTER ADDRESS;
18 2 = END RQACPT;
19 1 = RQISND:
    = PROCEDURE (IED$PTR) EXTERNAL;
20 2 = DECLARE IED$PTR ADDRESS;
21 2 = END RQISND;
    $include (common.elt)
22 1 = DECLARE TRUE LITERALLY '0FFH';
23 1 = DECLARE FALSE LITERALLY '00H';
24 1 = DECLARE BOOLEAN LITERALLY 'BYTE';
25 1 = DECLARE FOREVER LITERALLY 'WHILE 1';
    $include (resume.ext)
26 1 = RQRESM:
    = PROCEDURE (T$PTR) EXTERNAL;
27 2 = DECLARE T$PTR ADDRESS;
28 2 = END RQRESM;
    $include (:fl:iptask.elt)
29 1 = DECLARE IP$TASK$DESCRIPTOR LITERALLY 'STRUCTURE'
    = DELAY$LINK$FORWARD ADDRESS,
    = DELAY$LINK$BACK ADDRESS,
    = THREAD ADDRESS,
    = DELAY ADDRESS,
    = EXCHANGE$ADDRESS ADDRESS,
    = SP ADDRESS,
    = MARKER ADDRESS,
    = PRIORITY BYTE,
    = STATUS BYTE,
    = NAME$PTR ADDRESS,
    = TASK$LINK ADDRESS,
    = IP$THREAD ADDRESS);
30 1 declare
    rql4ex (15) byte external;
31 1 int$hnd: procedure public;
32 2 declare
    (msg$ptr, task$ptr) address;
33 2 do forever;
34 3 msg$ptr=rqwait (.rql4ex, 0);
35 3 call lock (7);
36 3 if int$proc$exch <> 0 then
37 3 do;

```



```

38 4 task$ptr = int$proc$exch;
39 4 int$proc$exch = 0;
40 4 end;
    else
41 3 call unlock(7);
42 3 call rqrsm(task$ptr);
43 3 end; /* of do forever */
44 2 end; /* of task */
45 1 end int$handler;

```

MODULE INFORMATION:

```

CODE AREA SIZE = 003DH 61D
VARIABLE AREA SIZE = 0004H 4D
MAXIMUM STACK SIZE = 0002H 2D
108 LINES READ
0 PROGRAM ERROR(S)

```

```

$title('Global Initialization, Version 1.1')
1 global$init:
do;

```

```

/* initialize semaphores */
+++++
do i=0 to 7;

```

PUBLIC PROCEDURE: GLOBAL\$INIT: LAST CHANGED 02/11/80

This is an initialization routine that is called by only one of the processors in the system. It initializes those structures that must be initialized before any IP routines are used and must then be left alone.

```

+++++
*/

```

```

$include(:fl:ipexch.elt)
2 1 = DECLARE IP$EXCHANGE$DESCRIPTOR LITERALLY "STRUCTURE(
= MSG$HEAD ADDRESS,
= MSG$TAIL ADDRESS,
= TASK$HEAD ADDRESS,
= TASK$TAIL ADDRESS,
= NEXT$EXCH ADDRESS,
= RESERVED (10) BYTE,
= IP$TASK$HEAD ADDRESS,
= IP$TASK$TAIL ADDRESS);
$include(:fl:global.ext)

```

```

+++++ /* Declaration of global inter-processor
data structures */

```

```

3 1 = declare
proc$pid byte external;
my$pid byte external;
in$que address external;
4 declare
my$proc$pid literally 'my$pid';
int$proc$exch literally 'in$que';
+++++

```

```

5  1  declare
      ip$exch$tab address external,
      ip$exch (1) ip$exchange$descriptor at(.ip$exch$tab),
      num$ip$exch byte external;
6  1  declare semaphore(8) byte external;

7  1  global$init: procedure public;

3  2  declare i byte;

      /* initialization the int$proc$exch queue */

9  2  int$proc$exch=0;
      /* initialize user ip$exch descriptors */
10 2  do i=0 to num$ip$exch-1;
11 3  ip$exch(i).ip$task$head=0;
12 3  ip$exch(i).ip$task$tail=.ip$exch(i);
13 3  end;

      /* initialize proc$id field */

14 2  proc$id=0FFFH;

      /* initialize semaphores */
+++++
15 2  do i=0 to 7;
16 3  semaphore(i)=0;
17 3  end;

18 2  return;
19 2  end;

20 1  end global$init;
+++++
MODULE INFORMATION:
CODE AREA SIZE = 0075H 117D
VARIABLE AREA SIZE = 0001H 1D
MAXIMUM STACK SIZE = 0002H 2D
73 LINES READ
0 PROGRAM ERROR(S)

1  $title('LOCK and UNLOCK procedures, Version 1.1')
   SEMA:
      do;

      /*
+++++
PUBLIC PROCEDURE: LOCK,UNLOCK. LAST CHANGED 10/12/79

These procedures provide mutual exclusion on access
to global data structures used in the interprocessor
communication package. Both routines use the seventh
semaphore for global data and the sixth semaphore for
interrupts. Software semaphores are used.
+++++

```

```

      */
      $include(exch.elt)
2   1  =  DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      =  MSG$HEAD ADDRESS,
      =  MSG$TAIL ADDRESS,
      =  TASK$HEAD ADDRESS,
      =  TASK$TAIL ADDRESS,
      =  NEXT$EXCH ADDRESS)';

3   1  rqwait: procedure(exch$ptr,delay$time) address external;
4   2      declare (exch$ptr,delay$time) address;
5   2      end rqwait;

6   1  s$mask: procedure(mask) external;
7   2      declare mask byte;
8   2      end s$mask;

9   1  declare timeout exchange$descriptor external;
10  1  declare semaphore(8) byte external;

/* lock procedure called upon entry of code modifying
   data structures */

11  1  lock: procedure (sema$num) reentrant public;
12  2      declare (sema$num,sema) byte;
13  2      declare (mem$ptr) address;

14  2      sema=1;
15  2      do while sema=1;
16  3          mem$ptr=rqwait(.timeout,1);
17  3          call s$mask(0C0H);
18  3          sema=semaphore(sema$num);
19  3          semaphore(sema$num)=0FFH;
20  3          call s$mask(040h);
21  3      end;
22  2      return;
23  2      end; /* of LOCK */

/* unlock called to exit region */

24  1  unlock: procedure(sema$num) reentrant public;
25  2      declare sema$num byte;
26  2      semaphore(sema$num)=0;
27  2      return;
28  2      end; /* of unlock */

29  1  end SEMA;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 005FH      95D
VARIABLE AREA SIZE  = 0000H      0D
MAXIMUM STACK SIZE  = 0006H      6D
67 LINES READ
0 PROGRAM ERROR(S)

```

```

$title('IP$INIT initialization routine')\
1  Ip$init:
    do;
    /*
    ++++++
    PUBLIC PROCEDURE: IP$INIT. LAST CHANGED 10/11/79-
    Initialization routine. Creates RQL4EX, initializes
    the interrupts and the interrupt handlers and creates
    the interrupt task int$hnd.
    ++++++
    */
    $include(:fl:global.ext)
    = /* Declaration of global inter-processor
    = data structures */
    2  1  = declare
    =         proc$id byte external,
    =         my$pid byte external,
    =         in$que address external;
    3  1  = declare
    =         my$proc$id literally 'my$pid',
    =         int$proc$exch literally 'in$que';
    $include(:fl:sema.ext)
    4  1  = lock: procedure (sema$id) external;
    5  2  = declare sema$id byte;
    6  2  = end;
    7  1  = unlock: procedure (sema$id) external;
    8  2  = declare sema$id byte;
    9  2  = end;
    $nolist
    34  1  int$hnd: procedure external;
    35  2  end int$hnd;
    36  1  set$vec$4: procedure external;
    37  2  end set$vec$4;
    38  1  declare
    =         int$hnd$stl literally '60',
    =         int$hnd$stk (int$hnd$stl) byte,
    =         int$hnd$std (20) byte,
    =         rql4ex (15) byte public,
    =         int$hnd$pri literally '65';
    39  1  declare int$hnd$std static$task$descriptor data(
    =         'int$hnd',
    =         .int$hnd,
    =         .int$hnd$stk,
    =         int$hnd$stl,
    =         int$hnd$pri,
    =         0,
    =         .int$hnd$std);

```

AP-88

```
40 1      ip$init: procedure public;

      /* wait for someone to perform global initialization
        and reset semaphores */

41 2      call lock(7);
42 2      call unlock(7);

43 2      output(0EBH)=80H;
44 2      output(0EBH)=0EH;
45 2      call rqcxch(.rq14ex);
46 2      call rqctsk(.int$hnd$std);
47 2      call rqsetv(.set$vec$4,4);
48 2      call rqelvl(4);
49 2      return;
50 2      end; /* of procedure */

51 1      end ip$init;
```

MODULE INFORMATION:

```
CODE AREA SIZE      = 003DH      61D
VARIABLE AREA SIZE = 005FH      95D
MAXIMUM STACK SIZE = 0002H      2D
133 LINES READ
0 PROGRAM ERROR(S)
```

APPENDIX B
BIPI PROGRAM LISTING


```

40      ip$init: procedure public;
      /* wait for someone to perform global initialization
      and reset semaphores */
41      call lock(V);
42      call unlock(V);
43      output(0E8H)=80H;
44      output(0E8H)=0EH;
45      call rdxchp(14);
46      call rdxchk(14);
47      call rdxstv(14);
48      call rdxlvi(4);
49      return;
50      end; /* of procedure */
51      end ip$init;

```

```

MODULE INFORMATION:
CODE AREA SIZE = 003DH
VARIABLE AREA SIZE = 002FH
MAXIMUM STACK SIZE = 0002H
133 LINES READ
0 PROGRAM ERROR(S)

```

APPENDIX B

BIPI PROGRAM LISTING

```

$TITLE('RMX/BASIC MULTIPROCESSOR INTERFACE, VERSION 1.1')
/*****
*
*      BIPI
*
*      THIS IS THE INTERFACE BETWEEN THE RMX/BASIC PROCESSOR AND
*      THE MULTI-PROCESSOR WORLD. THE RMX/BASIC INTERPORCESSOR
*      INTERFACE (BIPI) INTERFACES TO THE TERMINAL HANDLER AND TO
*      THE DISK IO SYSTEM. THE BIPI SUPPORT THE RECEIPT OF THE
*      FOLLOWING MESSAGE TYPES:
*
*      TERMINAL HANDLER REQUESTS:
*          READ$TYPE      = 8          READ FROM TERMINAL
*          WRITE$TYPE     = 12         WRITE TO TERMINAL
*
*      DISK I/O SYSTEM REQUESTS:
*          DFSS$RD        = 72         READ FROM DISK
*          DFSS$WT        = 76         WRITE TO DISK
*          DFSS$SK        = 77         DISK SEEK OPERATION
*          DFSS$CLS       = 78         DISK FILE CLOSE
*          DFSS$OPN       = 79         DISK FILE OPEN
*
*      BASIC TASK COMMAND REQUESTS:
*          SUSBAS         = 128        SUSPEND BASIC TASK
*          RESBAS         = 129        RESUME BASIC TASK
*
*      THE TASK WILL BE ENTERED BY SENDING THE NORMAL REQUEST MES-
*      SAGE TO THE GLOBAL EXCHANGE BIPI$EXCH.
*****/

```

```

1      BIPI$MODULE: DO;

      /* DEFINE THE LITERALS */

2      1      DECLARE READ$TYPE LITERALLY '8';
3      1      DECLARE WRITE$TYPE LITERALLY '12';
4      1      DECLARE DFSS$RD LITERALLY '72';
5      1      DECLARE DFSS$WT LITERALLY '76';
6      1      DECLARE DFSS$SK LITERALLY '77';
7      1      DECLARE DFSS$CLS LITERALLY '78';
8      1      DECLARE DFSS$OPN LITERALLY '79';
9      1      DECLARE SUSBAS LITERALLY '128';
10     1      DECLARE RESBAS LITERALLY '129';

      /* DECLARATION OF INTERNAL VARIABLES */

11     1      DECLARE (MSG$PTR,RESPONSE$PTR) ADDRESS;
12     1      DECLARE (TYPE) BYTE;

      /* DEFINE RMX PROCEDURES USED BY THE TASK */

13     1      RQSEND:
14           2      PROCEDURE (EXCH$PTR,MESSAGE$PTR) EXTERNAL;
15           2      DECLARE (EXCH$PTR,MESSAGE$PTR) ADDRESS;
16           1      END RQSEND;

17           1      RQWAIT:
18           2      PROCEDURE (EXCH$PTR,TIMEOUT) ADDRESS EXTERNAL;
19           2      DECLARE (EXCH$PTR,TIMEOUT) ADDRESS;
20           2      END RQWAIT;

```

```

19 1  RQSUSP:
20 2  DECLARE (TASK$DESC$PTR) EXTERNAL;
21 2  END RQSUSP;
22 1  RQRESM:
23 2  DECLARE (TASK$DESC$PTR) EXTERNAL;
24 2  END RQRESM;

/* DEFINE INTER-PROCESSOR PROCEDURES USED
   BY THE TASK */

25 1  IPSEND:
26 2  DECLARE (EXCH$PTR, MESSAGE$PTR) EXTERNAL;
27 2  END IPSEND;
28 1  IPWAIT:
29 2  DECLARE (EXCH$PTR, DUMMY) ADDRESS EXTERNAL;
30 2  END IPWAIT;
31 1  IPINIT:
32 2  END IPINIT;
33 1  SET$VEC$4:
34 2  END SET$VEC$4;

/* DEFINE THE EXCHANGES REQUIRED BY
   THE TASK */

$INCLUDE (:F1:EXCH.DEF)

35 1  = DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      = MESSAGE$HEAD ADDRESS,
      = MESSAGE$TAIL ADDRESS,
      = TASK$HEAD ADDRESS,
      = TASK$TAIL ADDRESS,
      = EXCHANGE$LINK ADDRESS )';
36 1  = DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      = MESSAGE$HEAD ADDRESS,
      = MESSAGE$TAIL ADDRESS,
      = TASK$HEAD ADDRESS,
      = TASK$TAIL ADDRESS,
      = EXCHANGE$LINK ADDRESS,
      = LINK ADDRESS,
      = LENGTH ADDRESS,
      = TYPE BYTE )';

37 1  DECLARE BIPIS$EX EXCHANGE$DESCRIPTOR EXTERNAL;
38 1  DECLARE RQINPX EXCHANGE$DESCRIPTOR EXTERNAL;
39 1  DECLARE RQOUTX EXCHANGE$DESCRIPTOR EXTERNAL;
40 1  DECLARE RQOPNX EXCHANGE$DESCRIPTOR EXTERNAL;
41 1  DECLARE BIPIS$RE EXCHANGE$DESCRIPTOR EXTERNAL;

/* DEFINE THE BASIC MESSAGE STRUCTURE OF
   INCOMING MESSAGES */

42 1  DECLARE MSG BASED MSG$PTR STRUCTURE (
      LINK ADDRESS,
      LENGTH ADDRESS,

```

AP-88

```

/* SUPPORT REQUESTS FOR 'BASIC' TASK OPERATIONS */
/* TYPE, BYTE, ADDRESS, HOME$EXCHANGE, RESPONSE$EXCHANGE, ADDRESS */
/* THEN DO */
/* DEFINE LOCATION OF 'BASIC' TASK DESCRIPTOR */
43 1 DECLARE BASC$TD ADDRESS EXTERNAL;
/* BEGINNING OF BIPI PROGRAM PROCEDURE */
44 1 BIPI: * RESUME TASKS INTO OPERATION *
PROCEDURE PUBLIC;
/* AT INITIALIZATION WAIT FOR COMMUNICATIONS
PROCESSOR TO COME UP */
45 2 MSG$PTR = RQWAIT(.BIPI$RE, 100);
46 2 CALL IP$INIT;
47 2 DO WHILE 1;
/* WAIT FOR A REQUEST FROM A PROCESSOR ON BUS */
48 3 MSG$PTR = IPWAIT (.BIPI$EX, 0);
/* SWAP RESPONSE ADDRESSES FOR LATER USE */
49 3 RESPONSE$PTR = MSG.RESPONSE$EXCHANGE;
50 3 MSG.RESPONSE$EXCHANGE = .BIPI$RE;
/* SAVE MESSAGE TYPE FOR TESTING */
51 3 TYPE = MSG.TYPE;
/* TEST FOR A READ REQUEST FROM TERMINAL */
52 3 IF TYPE = READ$TYPE
THEN CALL RQSEND (.RQINPX, MSG$PTR);
/* TEST FOR A WRITE REQUEST TO TERMINAL */
54 3 IF TYPE = WRITE$TYPE
THEN CALL RQSEND(.RQOUTX, MSG$PTR);
/* CONVERT REQUEST TYPE TO DISK COMPATIBLE TYPE */
56 3 IF TYPE > 71
THEN MSG.TYPE = MSG.TYPE - 64;
/* HANDLE OPEN DISK FILE REQUEST */
58 3 IF TYPE = DFS$OPN
THEN CALL RQSEND (.RQOPNX, MSG$PTR);
/* SUPPORT ALL OTHER REQUEST TYPES FOR DISK I/O */
60 3 IF ((TYPE > 71) AND (TYPE < 79)) THEN
CALL RQSEND (MSG.HOME$EXCHANGE, MSG$PTR);

```

```

/* SUPPORT REQUESTS FOR 'BASIC' TASK OPERATIONS */
62  3      IF TYPE > 127
          THEN DO;
          /* SUSPEND TASKS IN OPERATION */
64  4      IF TYPE = SUSBAS
          THEN CALL RQSUSP (BASC$TD);

          /* RESUME TASKS INTO OPERATION */
66  4      IF TYPE = RESBAS
          THEN CALL RQRESM (BASC$TD);
68  4      END;
          /* WAIT FOR RESPONSE FOR NON-'BASIC' OPERATIONS */

69  3      ELSE MSG$PTR = RQWAIT (.BIPI$RE, 0);

          /* SWAP RESPONSE EXCHANGES AGAIN BEFORE
          RETURNING MESSAGE */
70  3      MSG.RESPONSE$EXCHANGE = RESPONSE$PTR;
          /* RETURN THE MESSAGE TO THE CALLING PROGRAM */
71  3      CALL IPSEND (RESPONSE$PTR, MSG$PTR);

          /* END OF TASK */

72  3      END;
73  2      END BIPI;
74  1      END BIPI$MODULE;

MODULE INFORMATION:
CODE AREA SIZE      = 0101H      257D
VARIABLE AREA SIZE = 0005H      5D
MAXIMUM STACK SIZE = 0002H      2D
206 LINES READ
0 PROGRAM ERROR(S)

/* TEST FOR A WRITE REQUEST TO TERMINAL */
IF TYPE = READTYPE
THEN CALL ROSEND (.RQINPX, MSG$PTR);

/* TEST FOR A WRITE REQUEST */
IF TYPE > 17
THEN MSG$TYPE = MSG$TYPE - 8;

/* HANDLE OPEN DISK FILE REQUEST */
IF TYPE = DISKOPN
THEN CALL ROSEND (.RQOPNX, MSG$PTR);

/* SUPPORT ALL OTHER REQUEST TYPES FOR DISK I/O */
IF ((TYPE > 17) AND (TYPE < 19)) THEN
CALL ROSEND (MSG.HOMES$EXCHANGE, MSG$PTR);

```




APPLICATION NOTE

AP-109

Using Intel
Single Board Computers
for Serial Distributed
Processing Links

December 1980

2-175	INTRODUCTION
2-175	Common Network Designs
2-175	Loop Networks
2-175	Multidrop Networks
2-176	Star Networks
2-176	Sample Application

2-177	HARDWARE IMPLEMENTATION
2-177	Star Network Design
2-179	Multidrop Network Design

2-180	SOFTWARE IMPLEMENTATION
-------	-------------------------------

2-181	Master/Slave Relationships
2-181	Data Packet Formats
2-182	Multitasking Message Concepts
2-183	IRMX 80™ Named Exchanges
2-183	Extension
2-184	Generation of Multiprocessing Serial
2-184	Communications Link
2-184	Protocol Level Communications
2-184	Package
2-184	Link Level Communications
2-185	Package
2-185	Physical Level Communications
2-185	Package

2-185	APPLICATION EXAMPLE
-------	---------------------------

2-186	CONCLUSIONS
-------	-------------------

2-186	APPENDIX A — Extension to Name
2-186	IRMX™ 80 Exchanges

2-193	APPENDIX B — Master Communications
2-193	Software

2-202	APPENDIX C — Slave Communications
2-202	Software

2-215	APPENDIX D — Initialization Procedure
2-217	Queue Input
2-220	Queue Output
2-222	Queue to Queue Procedure
2-223	Host to Host Procedure
2-227	Check Point Procedure
2-230	Generate Host to Host Procedure
2-233	Generate Host to Host Procedure

2-236	APPENDIX E — Physical Link
2-236	for the iSBX 351™ Board

Using Intel Single Board
Computers for Serial
Distributed Processing Links

Peter L. Andersen
OEM Microcomputer Systems
Applications Engineering

Using Intel
Single Board Computers
for Serial Distributed
Processing Links

Contents

INTRODUCTION	2-175
Common Network Designs	2-175
Loop Networks	2-175
Multidrop Networks	2-175
Star Networks	2-176
Sample Application	2-176
HARDWARE IMPLEMENTATION	2-177
Star Network Design	2-177
Multidrop Network Design	2-179
SOFTWARE IMPLEMENTATION	2-180
Master/Slave Relationships	2-181
Data Packet Formats	2-181
Multitasking Message Concepts	2-182
iRMX 80™ Named Exchange	
Extension	2-183
Generation of Multiprocessing Serial	
Communications Link	2-184
Protocol Level Communications	
Package	2-184
Link Level Communications	
Package	2-185
Physical Level Communications	
Package	2-185
APPLICATION EXAMPLE	2-185
CONCLUSIONS	2-186
APPENDIX A — Extension to Name	
iRMX™ 80 Exchanges	2-189
APPENDIX B — Master Communications	
Protocol Software	2-193
APPENDIX C — Slave Communications	
Protocol Software	2-205
APPENDIX D —	
Queue Initialization Procedure	2-215
Queue Data Input Procedure	2-217
Queue Data Output Procedure	2-220
ASCII to Hex Conversion Procedure	2-222
Hex to ASCII Conversion Procedure	2-225
Checksum Calculation Procedure	2-227
Generate Ready Message Procedure	2-230
Generate Data Message Procedure	2-233
APPENDIX E — Physical Level Driver	
for the iSBX 351™ Board	2-236

INTRODUCTION

System designers are satisfying more and more difficult application needs with solutions which use microprocessors in a distributed environment. This distribution of intelligence results in many system benefits, including a simplification of the design and a resulting decrease in the development time of the system solution. An additional feature is the minimization of the installation costs of the system resulting from reduced wiring and from resource sharing. However, to effectively create distributed solutions, a reliable communication scheme must be used which links the various parts of the solution together.

The purpose of this application note is to demonstrate the ease with which Intel iSBC™ boards can be configured to implement distributed solutions. Several approaches are offered which apply standard operating modes of both the hardware and of the software. While certainly not the only solution, a distributed processor communications protocol is developed which will support many typical applications.

Distributed systems generally fall into one of two categories. The first consists of those systems in which the processors are located in such a manner as to allow communications over a common system data bus, such as Intel's MULTIBUS™ system bus. The second category involves systems in which the processors are also geographically distributed. This class of system generally uses a form of serial communications to allow each processor to communicate with other processors in the system.

The emphasis of this application note is on serial multiprocessor links. Before proceeding with a development of application solutions which involve serial communications, this note includes a short discussion of common network designs.

Common Network Designs

Serial networks may be classed into three general categories. These are the LOOP, the MULTIDROP, and the STAR. Each has applicability to certain iSBC products and has distinct advantages and disadvantages in an application. After discussing the characteristics of each network, an application scenario illustrates their uses.

LOOP NETWORKS

The loop represents the most common of the communication schemes. It is derived from the older teletype communication networks in which several printers were connected in series. Any data generated on one machine causes all machines concerned in the network to echo the data. This arrangement is shown in Figure 1. Loop communications normally use a 20-milliamp signal to convey the data. The presence of this current, known as

marking, represents a binary 1 and the absence, or spacing, of any circuit (an open circuit) represents a binary 0. Both full and half duplex configurations are commonly used.

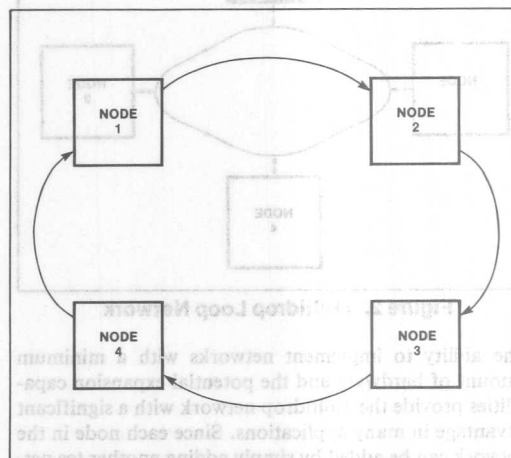


Figure 1. Serial Loop Configuration

The incorporation of loop networks into a system implementation creates both advantages and disadvantages. The strongest argument for using a loop is the inherent noise immunity which can be gained by using the 20-milliamp current to convey the data. The use of a high compliance voltage also allows the transmission of data for large distances at low to moderate transmission rates.

Solid state circuits common to current technology are somewhat less reliable than older mechanical units when many stations are configured onto the loop. This is the result of the requirement to reproduce the information at each node for transmission to the next node. If any node fails, all other nodes downstream in the communication network will not be capable of communicating. Even with this constraint, many good designs incorporate the loop network concepts.

MULTIDROP NETWORKS

Multidrop networks are fast becoming the accepted network for multiprocessor communications. This type of network is shown in Figure 2. As can be seen, this arrangement is similar to that of the loop. The advantage is that all stations are capable of receiving data simultaneously. Multidrop networks are voltage driven since to pass current through a node creates a loop situation. The electrical interface for multidrop networks consists of tri-state drivers and high impedance receivers. The RS422 electrical interface is common for this type of network since it provides high speed data transmission over long distances with good noise immunity.

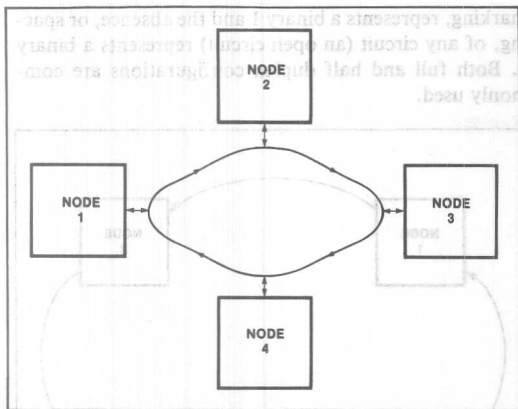


Figure 2. Multidrop Loop Network

The ability to implement networks with a minimum amount of hardware and the potential expansion capabilities provide the multidrop network with a significant advantage in many applications. Since each node in the network can be added by simply adding another tee network, expansion is performed without the need for modification of the communication network.

Intel's iSBX 351™ Serial MULTIMODULE™ Board is an ideal vehicle for the implementation of multidrop networks. Its use is detailed in subsequent paragraphs of this application note.

STAR NETWORKS

Star networks are implemented where either data throughput or hardware limitations prohibit the use of other types of communication arrangements. Figure 3 illustrates a typical star network implementation. Note that a star network is really an example of multiple point to point communications. System throughput is improved since communication can occur with all slave nodes simultaneously without the need to have each node monitor traffic which is not intended for it.

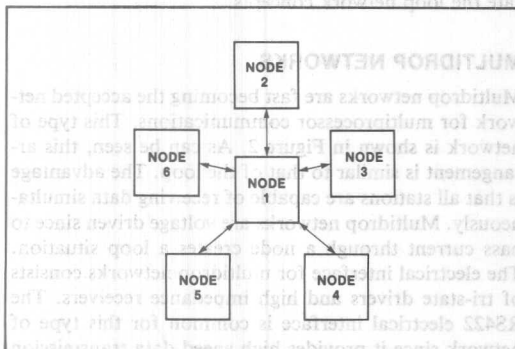


Figure 3. Star Configuration

Intel's iSBC 544™ Intelligent Communications Controller and the iSBC 534™ Four-Channel Communications Board are ideal choices to design star communications networks. The outer points of the star are implemented using any single board computer which has on-board serial communications. An example of this type of network is included in this application note.

Sample Application

An application example is used in this application note to illustrate the techniques which are required to implement various types of serial communication networks. Both hardware and software aspects of the design are described in some detail.

The application discussed in this note involves the creation of an alarm and security system for a multiple building complex. This complex is shown in Figure 4. The solution generated allows monitoring of three existing buildings with provision for the addition of a fourth at a later date. The figure shows that each building is to have a local annunciator panel for reporting activity in those areas served by the building sensors. In addition, a main security station provides monitoring of all buildings in the complex.

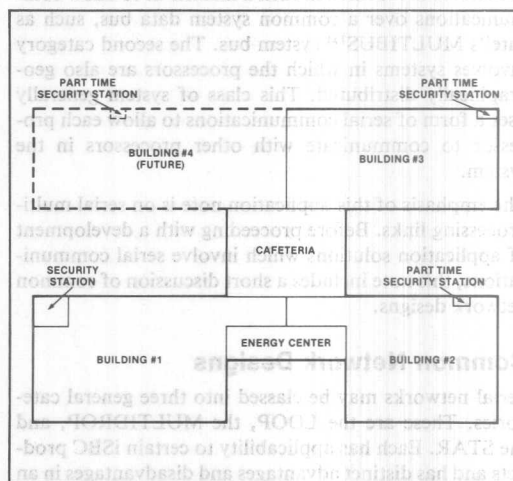


Figure 4. Application Building Complex

Applications such as this are optimized by distributing the system intelligence throughout the complex. This serves two needs; that of minimizing the wiring costs, and of providing an improvement in system reliability. When a system is distributed in this way, a means must be devised which allows communication between the various elements of the network. Both hardware network design and software communication protocol are required before the system can be considered operational.

In order to illustrate various network designs, the example is broken into a distributed system as shown in Figure 5. This is an example of a multidrop communication network. The system design does not require that building nodes communicate with each other; all communications are between building annunciator nodes and the main operator station. This illustrates the concept of a master and a slave. This will be further discussed in a later portion of this application note.

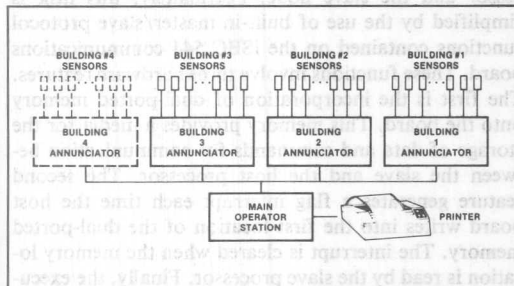


Figure 5. A Multidrop Application

The building annunciator can be further distributed as shown in Figure 6. Here, a star communication network is created which places data gathering intelligence near clusters of sensors. These stations, or nodes, then transmit information regarding local activity to their master node, the building concentrator/annunciator panel. Note that this panel serves both as a master node for the sensor units and as a slave node for the main operator station. This arrangement is common in many system designs.

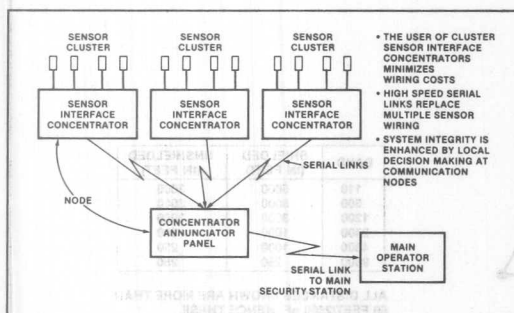


Figure 6. A Star Application

HARDWARE IMPLEMENTATION

The implementation of communication networks using iSBC boards for both star and multidrop techniques is detailed in the following sections. In each case, a master/slave relationship is assumed. For purposes of illustration, actual data from the application example is used in performing the sample calculations. While not

the only possible system designs, those shown are considered to be typical and should enable the reader to gain an insight into techniques which can be used to create similar designs.

Star Network Design

A star network can use almost any accepted transmission media. This is because each node communicates with only one other node, creating a multiple point to point link. Since the techniques are identical for all media, this note discusses only a representative example. RS232C data transmission is used to demonstrate the techniques which can be used to establish a star network.

The heart of a star network is the master node. It must provide a means of independent communication with each of its slave nodes. Costs can be minimized if multiple communication drivers are placed onto the same board. The example chosen requires that four slave nodes be controlled by the master. The iSBC 544 Intelligent Communications Controller provides this function. It provides a software selectable baud rate and the ability to offload the host processor of communication activities. If additional slave nodes are required, the system can be expanded by adding iSBC 544 controller boards.

An ideal choice for a slave node is the iSBC 80/10B™ Single Board Computer. This computer provides good processing capabilities with low cost. It includes both an RS232C and 20-milliamp current loop communications interface. For many small applications it makes an ideal choice for a complete single board solution.

Figure 7 shows the implementation for a four-node plus master star communication network for the security application. Both the iSBC 544 communications board and the iSBC 80/10B Single Board Computer require jumpering and component insertion to allow operation in either the RS232C mode or in the EIA mode. EIA operation is electrically identical to RS232C but has specifications which allow transmissions at significant distances when the baud rate is reduced. For the purposes of this application note, the terms are used interchangeably.

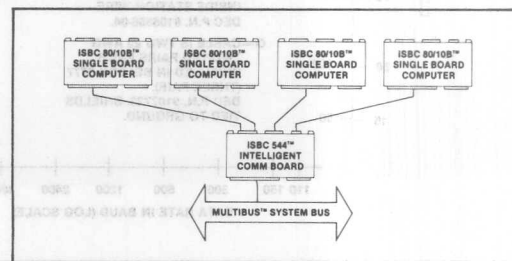


Figure 7. Star Implementation

In the case of the iSBC 544 board, a header plug can be inserted into an on-board connector to enable the correct mode of operation. The wiring for this header is shown in Figure 8. Note that the ready to send and the clear to send signals are jumpered to allow the correct operation of the USART without the control signal of a modem. Also note that the transmit and receive signals are reversed through the header.

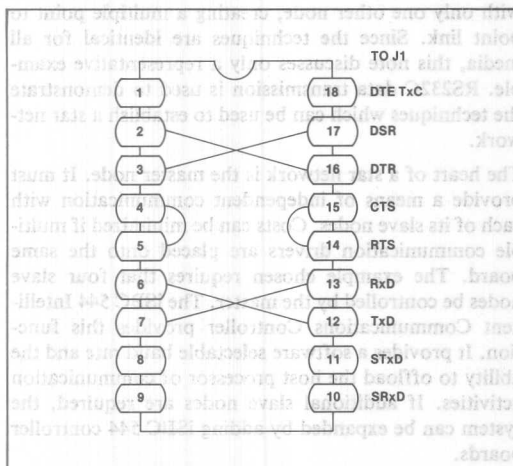


Figure 8. Master to Master Wiring Header.

The use of an RS232C signal for long distance communication necessitates the use of a low baud rate for reliable data transmissions. This is shown in Figure 9.

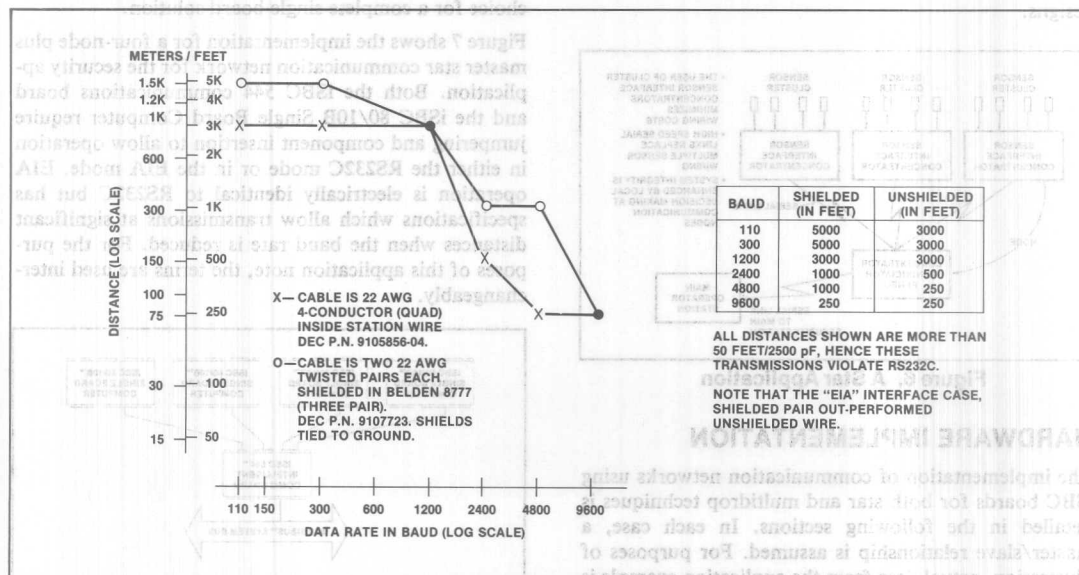


Figure 9. RS232C Baud Rate vs Distance

The security system application design allows for minimum communication traffic so a baud rate of 1200 is used for the star networks. The placement of intelligence at the sensor cluster minimizes the amount of data which must be transmitted, providing adequate response time at this data rate.

The hardware design phase includes the selection of the communication scheme to be used between the host processor and the slave node. Fortunately, this task is simplified by the use of built-in master/slave protocol functions contained on the iSBC 544 communications board. These functions involve three hardware features. The first is the incorporation of dual-ported memory onto the board. This memory provides a media for the storage of data and commands for communication between the slave and the host processor. The second feature generates a flag interrupt each time the host board writes into the first location of the dual-ported memory. The interrupt is cleared when the memory location is read by the slave processor. Finally, the execution of the 8085A-2 SOD instruction generates a MULTIBUS interrupt to inform the host board of the need to service the slave. This interrupt can be vectored to the host interrupt matrix to activate a service routine in support of the slave board.

A detailed explanation of the use of these features is found in another application note, AP-53, *Using the iSBC 544™ Intelligent Communications Controller*. The reader should refer to this document for details of the board and its use.

Multidrop Network Design

The complexity of a multinode system is minimized through the use of a multidrop network. Certain limitations are placed on the acceptable hardware transmission media in multidrop applications. A scheme must be used which allows each of the nodes to alternatively gain control of the network in order to communicate its message.

Both half and full duplex configurations are allowable in a multidrop network. While it is simpler from a hardware standpoint, a half duplex connection requires more stringent communications protocol as this system allows communications in only one direction at a time. For all practical purposes, no priority for masters or slaves is provided. All nodes may listen to whomever is using the line at the time. The software must be written to allow only one node to communicate at a given instant. An example of a half duplex network is the Ethernet.

More straightforward software can be written for full duplex communication configurations. In this instance,

an assumption is made that only one master is configured into the system and always drives the output lines. Any number of slaves can be placed to drive the input lines, communicating only when queried by the master node. This is the scheme used for the application example which is discussed in this application note.

Intel's iSBX 351 Serial MULTIMODULE Board lends itself well to a multidrop application. It allows a wide variety of system solutions to be created using any of the single board computers which incorporate iSBX bus connectors. Several board options must be enabled to create the multidrop communications configuration.

The first option involves configuring the drivers to use a tri-state driver on the output lines. This allows only the board desiring to control the communications signals to place data on the serial output lines. The board configuration is different for a slave and for a master node. In the Intel implementation, the driver output signal lines are controlled using the DTR (data terminal ready) signal from the USART. The jumper configuration for both a slave and a master is shown in Figures 10 and 11.

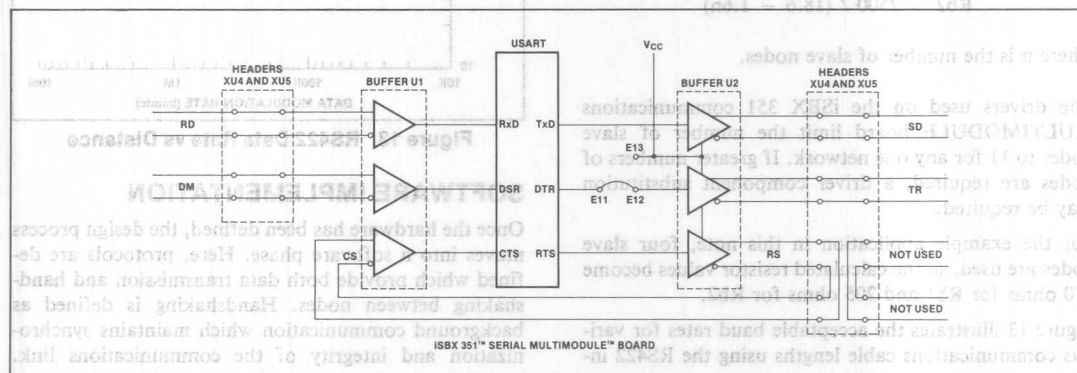


Figure 10. Master Configuration Detail

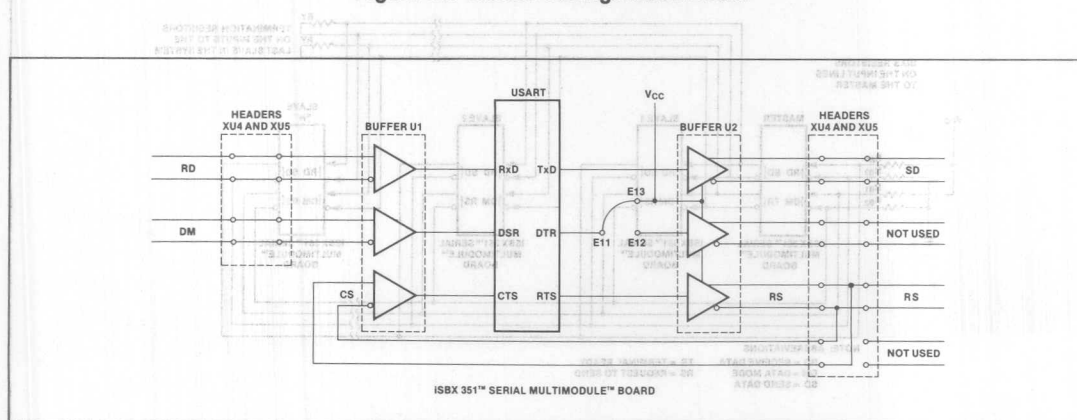


Figure 11. Slave Configuration Detail

Figure 12 illustrates the physical wiring data paths which are used to create a full duplex multidrop network having a master and four slave stations. Each board must be configured using a header block to provide the correct signals. This is done as shown in Figures 10 and 11.

Finally, bias resistors are chosen to terminate the serial communication data paths. A discussion of the formulas for determining the correct values for the two resistors is provided in Appendix A of the *iSBX 351™ Serial MULTIMODULE Board Hardware Reference Manual* (Order Number 9803190). If the equations are solved for the components used on the RS422 section of the board, the equations for calculating the correct values become:

$$Rb1 = 4500 / (18.6 - 1.6n)$$

where n is the number of slave nodes

AND

$$Rb2 = 2500 / (18.6 - 1.6n)$$

where n is the number of slave nodes.

The drivers used on the iSBX 351 communications MULTIMODULE board limit the number of slave nodes to 11 for any one network. If greater numbers of nodes are required, a driver component substitution may be required.

For the example application in this note, four slave nodes are used, so the calculated resistor values become 370 ohms for Rb1 and 205 ohms for Rb2.

Figure 13 illustrates the acceptable baud rates for various communications cable lengths using the RS422 in-

terface. The iSBC 351 board is designed to support data transmission rates of up to 19.2 kilobaud. Thus, the system designed using these boards is seen to permit a total multidrop cable length of up to 4000 feet (1.219 km). The sample application discussed in this note incorporated a total communications link length of 1600 feet, well within the performance limits.

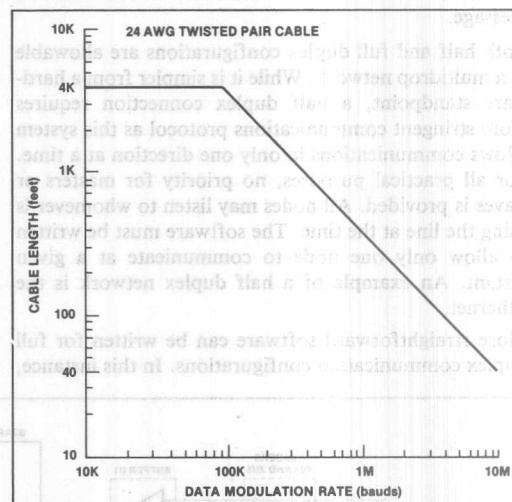


Figure 13. RS422 Data Rate vs Distance

SOFTWARE IMPLEMENTATION

Once the hardware has been defined, the design process moves into a software phase. Here, protocols are defined which provide both data transmission and handshaking between nodes. Handshaking is defined as background communication which maintains synchronization and integrity of the communications link.

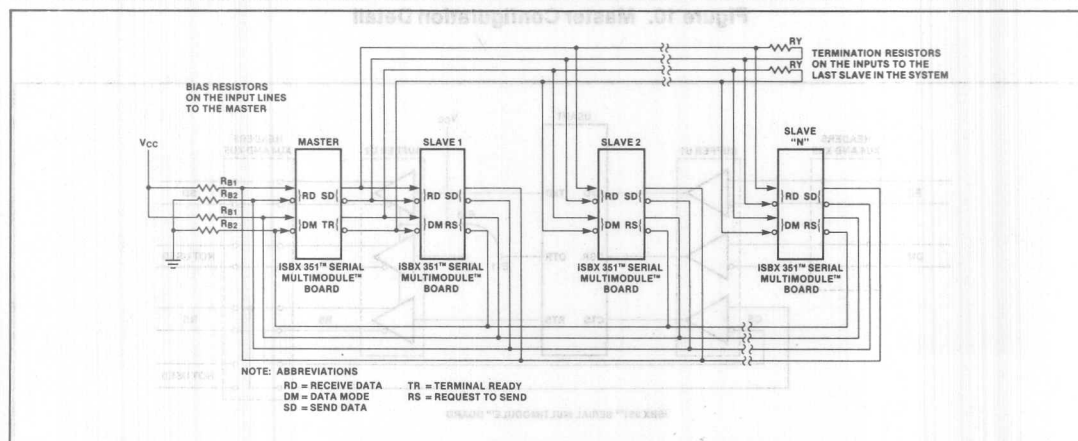


Figure 12. Master/Slave Physical Wiring

Again, standard features of both iSBC board products and of iRMX™ software products simplify the incorporation of communication protocols into the system solution.

Master/Slave Relationships

In order to maintain an orderly flow of data between nodes, software must be created which allocates priority and only allows one node to transmit at any instant. The most straightforward technique which incorporates this function is the master/slave relationship. Here, one node is designated as a master and all others are slaves. Each slave is allocated a time during which it may transmit any information to other nodes. The master node controls the prioritization of all slave nodes. The complexity of the system can be further reduced if a full duplex scheme is implemented. In this mode, all slaves listen only to the master and transmit their messages over a common serial channel to the master. A disadvantage of this technique is that direct slave to slave (virtual channel) transmissions are not permitted.

Figure 14 shows how a master node establishes time slots for windows, during which a slave may control the output channel and transmit data to the master. Each slave node is assigned a unique identification or address.

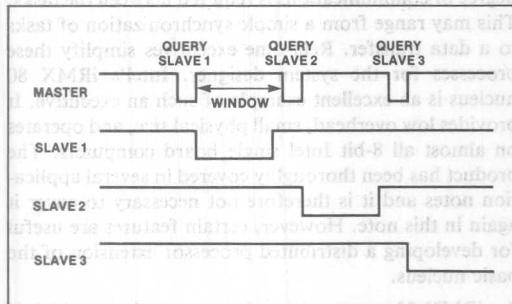


Figure 14. Generation of Windows

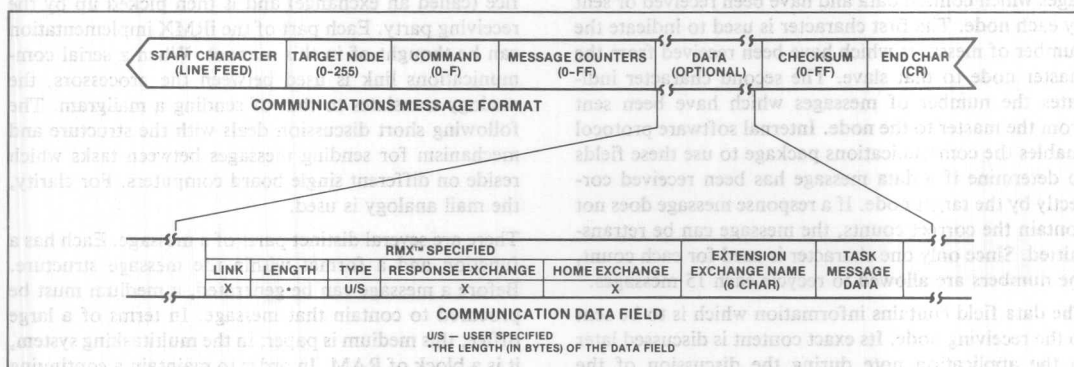


Figure 15. Communications Packet Format

Each data packet contains an address as an integral part. When a slave recognizes its address in the packet from the master, it knows that it has fixed time window in which to return its own data packet to the master.

Data Packet Formats

Communications between each slave and its master involve the transmission of data packets. Each packet contains both handshaking information (such as checksums and message counts) and the information which is to pass between nodes. The handshaking portions of the message are used to maintain the integrity of transmissions in the face of such external stimuli as electrical noise.

Debugging and system maintenance is minimized when a means is provided which allows easy viewing and interpretation of the communication messages. The scheme used in this application note uses a fully ASCII compatible communications format. This allows a CRT terminal to tap onto the link and to monitor all communication messages.

Each message packet begins and ends with a unique character. This character cannot exist with a message block. Using the ASCII formats, all messages use the numeric representation of 0 to 9 and the alpha characters from A to Z. A line feed is used to begin a message packet and a carriage return is used to terminate the message. The assignment of these characters assures an easily interpreted message packet.

The layout of the data packet is shown in Figure 15. The justification and description of each field is given in the following paragraphs. Note that much of the message is concerned with maintaining system integrity. The requirement for an address field has already been described. Two characters are used for this field and provide for 256 addresses (0-FF hex). The protocol being described used 0 as the system master and 1 through 255 for possible slave addresses.

The next field indicates the type of message being sent and the action which is required by the receiver. Analysis of system communication requirements indicate that five message types are sufficient. The use of a single byte of the packet for this field allows a total of 16 message types. This leaves 11 for future expansion. The types being used in this application are:

Type 0 — This is a reset request from the master to a slave. It will cause the target slave to generate a software reset. This command is used only when communications cannot be established using other means.

Type 1 — A type 1 command is used to synchronize the master and the slave. It will cause the message counters (see subsequent paragraphs) to reset to zero. This command is issued only by the master node.

Type 2 — The type 2 message is defined to be that which contains information which is to be moved between nodes of the communications network. The exact format of the data is discussed elsewhere. This is the only message which is not considered to be of a strictly handshaking nature.

Type 3 — A type 3 message is used to indicate that a network node is responsive and ready to handle messages. When normal communications have been established and no data messages are being sent, this message is continuously being sent between the master and the slaves.

Type 5 — The final message type is the type 5. It is used by the slave to respond to the master's request to synchronize. The receipt of this message indicates that the slave has performed all necessary operations with its message counters and is ready to receive messages.

The next field is used to describe the number of messages which contain data and have been received or sent by each node. The first character is used to indicate the number of messages which have been received from the master node to that slave. The second character indicates the number of messages which have been sent from the master to the node. Internal software protocol enables the communications package to use these fields to determine if a data message has been received correctly by the target node. If a response message does not contain the correct counts, the message can be retransmitted. Since only one character is used for each count, the numbers are allowed to recycle each 15 messages.

The data field contains information which is to be sent to the receiving node. Its exact content is discussed later in the application note during the discussion of the iRMX 80 message extensions.

A checksum is included to guarantee the integrity of the transmitted message. This field contains the 8-bit modulo 256 sum of all transmitted ASCII characters, beginning with the line feed and continuing through the last character of the data field. It is used to compare a calculated checksum of received characters with that transmitted by the sending node. If these two numbers are in agreement, the message is considered to be valid and can be used by the receiving node's application software.

The final field is the end of message character. A carriage return is used and provides a unique indicator of the end of a message packet.

The implementation of these communication concepts into a functioning software package is illustrated later during the discussion of the layering of the multiprocessing communications package.

Multitasking Message Concepts

The design of systems which involve distributed processors strongly suggests that a multitasking environment is also present. The ability to segregate an application into tasks allows a considerable simplification of the design and implementation process. In reality, few tasks represent completely isolated functions. Some degree of communications is required between the tasks. This may range from a simple synchronization of tasks to a data transfer. Real-time executives simplify these processes for the system designer. Intel's iRMX 80 nucleus is an excellent example of such an executive. It provides low overhead, small physical size, and operates on almost all 8-bit Intel single board computers. The product has been thoroughly covered in several application notes and it is therefore not necessary to cover it again in this note. However, certain features are useful for developing a distributed processor extension of the basic nucleus.

An iRMX 80 message is analogous to a letter which is mailed to someone. It is sent via a mailroom or post office (called an exchange) and is then picked up by the receiving party. Each part of the iRMX implementation can be thought of in this manner. When a serial communications link is used between the processors, the analogy translates to that of sending a mailgram. The following short discussion deals with the structure and mechanism for sending messages between tasks which reside on different single board computers. For clarity, the mail analogy is used.

There are several distinct parts of a message. Each has a function and a format within the message structure. Before a message can be generated, a medium must be procured to contain that message. In terms of a large office, this medium is paper; in the multitasking system, it is a block of RAM. In order to maintain a continuing supply of memory, the iRMX executive provides the

user with what is known as a free space manager. The purpose of the free space manager is to recycle memory blocks so that they can be again used for generation of additional messages. This service is called each time the sending task requires to generate a message. When the communication programs have successfully transmitted the message to the target node, the memory block is again returned to the free space memory pool. As the target node receives the message it must obtain a block of memory from its free space manager to store the information until it can be processed by the target task. The target task returns the block to the pool when it has taken the appropriate actions.

As with any type of message, the individual to whom the message is to be delivered must be provided. Because the iRMX 80 nucleus is designed to be used by multiple tasks on the same processor, the target address mechanism is not included in the message header itself; instead, the target is specified in the call to the iRMX procedure as a parameter list component. When the target exchange address is not known, which happens when multiprocessor applications are created, it is necessary to create an extension to the executive which allows the assignment of a target exchange name to each message. Fortunately, this is relatively easy to create.

iRMX 80™ Named Exchange Extension

The iRMX 80 nucleus uses a small block of RAM memory to store data regarding the characteristics and status of each exchange. This storage area normally occupies 10 bytes of memory. Its layout is shown in Figure 16. Except to note that a unique identification field does not exist which sets each exchange descriptor apart, it is beyond the scope of this application note to dwell on the meanings of the fields. However, a method must be created which allows the extension of the field to include an identification which is unique to that descriptor.

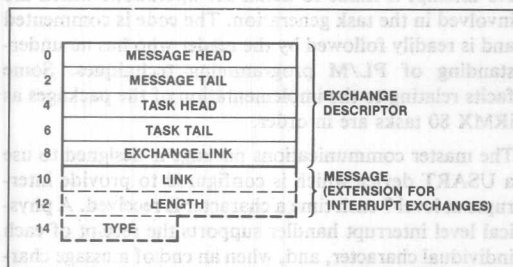


Figure 16. Exchange Descriptor

If one is not to rewrite the nucleus, the format of the exchange descriptor must not be changed and the identification field should be added as additional bytes of the memory block. Two features of the iRMX 80 nucleus make the direct implementation of this concept impossible. First, a special exchange descriptor already exists

within the standard real-time executive. This is the interrupt exchange which adds 5 bytes to the nucleus for use as an interrupt message. Second, if an additional extension were to be added, all exchanges would have to be created as interrupt exchanges so that common software could be used. While this would not in itself be prohibitive, the interactive configurator (ICU 80) does not allow addition of fields to either the standard or to the interrupt exchange descriptor. Another method is required to add the extension.

Fortunately, another method exists to create an exchange. The nucleus operation, RQCXCH, creates an exchange at an address which is specified by a parameter of the call instruction. If user software is created to build the named exchanges, a name can be prefixed to each desired name exchange.

In the standard iRMX 80 nucleus six characters are allowed to name each task. A logical extension of this concept provides a six-character name for each named exchange. The exchange descriptor is thus extended by inserting 6 bytes before the basic format. In reality, this is not sufficient because named exchanges are to be created dynamically. The memory blocks representing the set of named exchanges to not necessarily constitute a contiguous block of RAM. This leads to the creation of an additional word of memory to carry a pointer to the next named exchange field. A value of zero in this field indicates that no additional named exchanges exist. If a non-zero value is placed into the field, it is a pointer to the next named exchange. Figure 17 illustrates the structure of the named exchange fields.

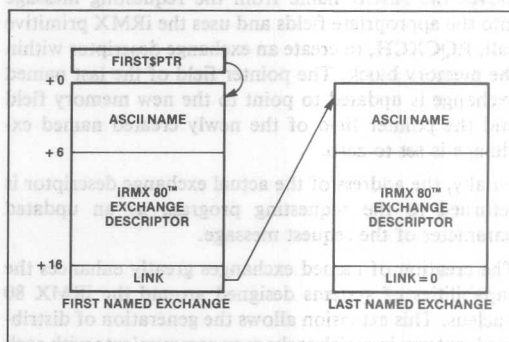


Figure 17. Named Exchange Descriptors

The creation of the software to support the named exchange extension is rather straightforward. Two functions are required; one which creates the named exchanges, and a second which will return the address of an exchange which matches a given name. A complete listing of the software generated to perform these functions is shown in Appendix A. The services of the free space manager are used to allocate the memory segments which are used for the exchange descriptors. A

quick examination of the program techniques provides some insight into the operations involved in creating extensions to the standard nucleus.

Examination of the listing at line 55 shows that a procedure is used to allow user code to determine the absolute address of the iRMX exchange descriptor field from the named exchange lists. This address can in turn be used with either the RQWAIT or the RQSEND instruction in the user's code. If no corresponding name is found in the table, a value of zero is returned by the procedure. An address parameter is passed in the user call to the FIND\$EXCH subroutine which points to the 6 bytes containing the ASCII name of the referenced exchange.

An internal address value, identified as FIRST\$PTR, contains the address of the first named exchange descriptor. A simple DO loop sequence is used to locate a matching name in the table lists. Either a zero or the location of an actual 10-byte exchange descriptor within the extended memory block is returned.

A task is used to provide the system capability of building named exchanges. A task, rather than a procedure, is included because a task allows the initialization of system pointers such as FIRST\$PTR. The named exchange building task, CREATE\$COM, waits at its input exchange, COM\$CREATE\$EXCH until a request for creation of a named exchange is received from a user task. The listing for this task is found in Appendix A, beginning on line 78.

When a message is received, the task obtains a 20-byte block of memory from the free space manager. It then moves the ASCII name from the requesting message into the appropriate fields and uses the iRMX primitive call, RQCXCH, to create an exchange descriptor within the memory block. The pointer field of the last named exchange is updated to point to the new memory field and the pointer field of the newly created named exchange is set to zero.

Finally, the address of the actual exchange descriptor is returned to the requesting program as an updated parameter of the request message.

The creation of named exchanges greatly enhances the capabilities of systems designed around the iRMX 80 nucleus. This extension allows the generation of distributed systems in which tasks may communicate with each other regardless of their geographical locations so long as some type of link exists.

Generation of Multiprocessing Serial Communications Link

The creation of software for a serial communications link provides the capability of allowing true multitasking, multiprocessor capabilities. The need for two types of communications packages, a slave and a master, indicates that two separate tasks are required. A compre-

hensive examination of the communication requirements indicates that the system can be generated in three layers. Figure 18 shows the layer relationships. The first is defined as the protocol and consists of that code which is required to implement the algorithms for either the slave or the master network nodes. The second level is known as the link level and contains that code which is used to support common operations such as message generation and data queue handling. The third provides a specialized interface to the physical hardware which is being used to generate the communications link. This level, called the physical level, is unique to each configuration.

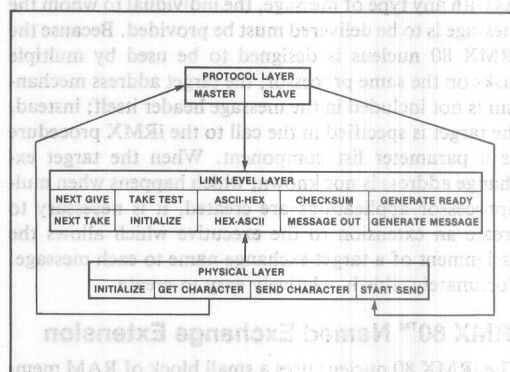


Figure 18.

PROTOCOL LEVEL COMMUNICATIONS PACKAGE

Two protocol level communications packages are included in this application note. One implements the master protocol and is reproduced in Appendix B. The second, the slave protocol, is found in Appendix C.

No attempt is made to detail the operations which are involved in the task generation. The code is commented and is readily followed by the reader who has an understanding of PL/M programming techniques. Some facts relating to the implementation of the packages as iRMX 80 tasks are in order.

The master communications package is designed to use a USART device which is configured to provide interrupts at level 7 each time a character is received. A physical level interrupt handler supports the receipt of each individual character, and, when an end of message character (carriage return) is received, places an interrupt message into the interrupt exchange, RQL7EX. At this point (Appendix B, line 244), the master protocol handler can begin processing the received message. Because the task is associated with interrupt level 7, a task priority in the range between 113 and 128 must be assigned to the task. The example used in this application note uses a priority of 113.

In a similar manner, the slave communications task uses interrupt level 6 to receive messages (Appendix C, line 130). Its priority must lie in the range between 97 and 112.

LINK LEVEL COMMUNICATIONS PACKAGE

The link level communications package provides a common set of procedures which can be used to support both the protocol and the physical layers. Listings of these support programs are found in Appendix D. The programs which make up the package are defined in the following discussion.

Several procedures support the maintenance of the data queues. These are CQ\$Q\$INIT, CQ\$NEXT\$TAKE, and CQ\$NEXT\$GIVE. The first, CQ\$Q\$INIT, is used to clear space for a data queue. It sets up the length parameter and the give and take pointers to their initial values.

Data is placed onto the queue by calling the procedure, CQ\$NEXT\$GIVE, and including the desired data in the paramter list. Conversely, CQ\$NEXT\$TAKE is used to take data from the queue. Both maintain the queue pointers as data is inserted or removed. For further information about data queues, the reader should refer to AP-52, *Using the iSBC 544™ Intelligent Communications Controller*. This document contains an extensive discussion on the subject of data queues.

Two procedures deal with the transformation of data between printable ASCII and the internal binary format. CQ\$ASC\$HEX transfers data from the data queue into a working buffer. In the process, it converts the format from ASCII into a hex representation usable by the target task. Likewise, CQ\$HEX\$ASC is used to transform data in a user buffer into a transmittable format. In both cases, appropriate start and stop characters are added or deleted as necessary to create the correct format.

One procedure deals with computing the checksum of a message which is in ASCII format. This procedure, CQ\$CHECKSUM, returns a zero if the checksum agrees with that in the message. If an error is indicated, a value of -1 (OFF hex) is returned.

Finally, two procedures support the generation of message packets. One, CQ\$GEN\$RDY, is used to build a "ready" message by creating the appropriate data into the fields. The second, CQ\$GEN\$MSG, generates a data message containing an iRMX 80 message to be sent to an exchange on another processor board.

PHYSICAL LEVEL COMMUNICATIONS PACKAGE

The physical level communications package provides the customization for the unique configuration of host

processor and USART. Four procedures must be included with this level. These are:

- **CQ\$INIT** — This public procedure contains all operations necessary to initialize the timers, counters and USARTs associated with the communications code. In addition, this procedure defines the interrupt service routines for the USART and enables the corresponding interrupt levels.
- **CQ\$START\$MSG** — A public procedure which places the USART into a mode in which the transmitter is enabled. The execution of this procedure should result in an interrupt being generated by the USART transmitter ready line.
- **CQ\$MIVT** — This is an interrupt service routine associated with the USART receiver ready line. It is entered each time that a character has been received by the communication node. In the case of a slave node, this procedure is known as CQ\$SIVT. The name is unimportant because the location of the routine is passed to the iRMX 80 nucleus at initialization by the CQ\$INIT program. When a carriage return (end of message) is encountered, a message is passed to the protocol level task by passing a message to the interrupt exchange, RQL6EX, using the iRMX primitive, RQISND.
- **SEND\$CHAR** — Like the CQ\$MIT routine, this is an interrupt handler procedure. It is entered each time the USART signals that it is ready to transmit a character. A new character is obtained from the data queue and it is transmitted to the receiving node. If the character is the end of message, flags are set and the USART transmitter is disabled.

The listing of a sample physical driver is provided in Appendix E. This driver illustrates the use of the iSBX 351 Serial MULTIMODULE Board placed into a socket of an iSBC 80/24 Single Board Computer.

The example from the application implements a multi-drop slave node. The enabling of the tri-state drivers is accomplished in line 138 by sending the USART a command of 025H. When the message has been sent, the driver is again placed into a high impedance mode by sending a command of 026H (line 121). These commands control the driver by toggling the DTR or Data Terminal Ready lines of the 8251A USART device.

APPLICATION EXAMPLE

The alarm and security system previously discussed provides a perfect environment in which to use the concepts developed in this application note.

To verify the functionality of the communications package, the transmissions between a master and a slave were monitored using a CRT terminal. The terminal was

connected to one of the RS232 serial paths using a tee network. This tee network is shown in Figure 19. Several scenarios were set up to test the effectiveness of the communications protocol. The results of some of these tests are recorded in the following discussion.

A test of normal communications was performed in which one message is passed between the master and the slave. A short time later, a message is passed to the master from the slave. The CRT display for this action displayed as:

```
(line 1) 0330000
(line 2) 00300FD
(line 3) 03200091230900BF0000000009F
(line 4) 00310FE
(line 5) 0331001
(line 6) 0021082470900870000000008A
(line 7) 0331102
(line 8) 00311FF
```

Line 1 is a "ready" (character 3) message from the master node to a slave addressed 03 hex (characters 1 and 2). No messages have been sent in either direction at the time this message was generated. On line 2, the slave has responded indicating that it is present and ready to receive messages. It also has not transmitted any messages to the master nor has it received any.

The master node sends the slave a message on line 3. Note that the message count field (characters 4 and 5) is

not changed until after the message has been transmitted. The message sent is a type BF and has a total length of 9 bytes. On line 4, the slave acknowledges that it has received one message and has sent none.

Lines 5, 6, and 7 illustrate a sequence in which the slave sends a message to the master node. Prior to the message being sent, the message count field shows that one message has been received. After receiving the message, the master node increments its received counter.

Tests with the alarm and security system verify that the communication module functions correctly. Data integrity is maintained even through intentional disruption of the electrical link. Both the RS232 and RS422 communication paths function as designed.

A detailed discussion of this application and of the multitasking capabilities of the iRMX 80 nucleus can be found in AP-112, *Simplifying Complex Designs Using The iRMX 80™ Executive*.

CONCLUSIONS

This application note illustrates the ease with which a user can add extensions to the iRMX 80 executive to support a distributed multiprocessing application. In addition, the user of standard "off the shelf" single board computers and expansion modules to create a hardware solution is explained.

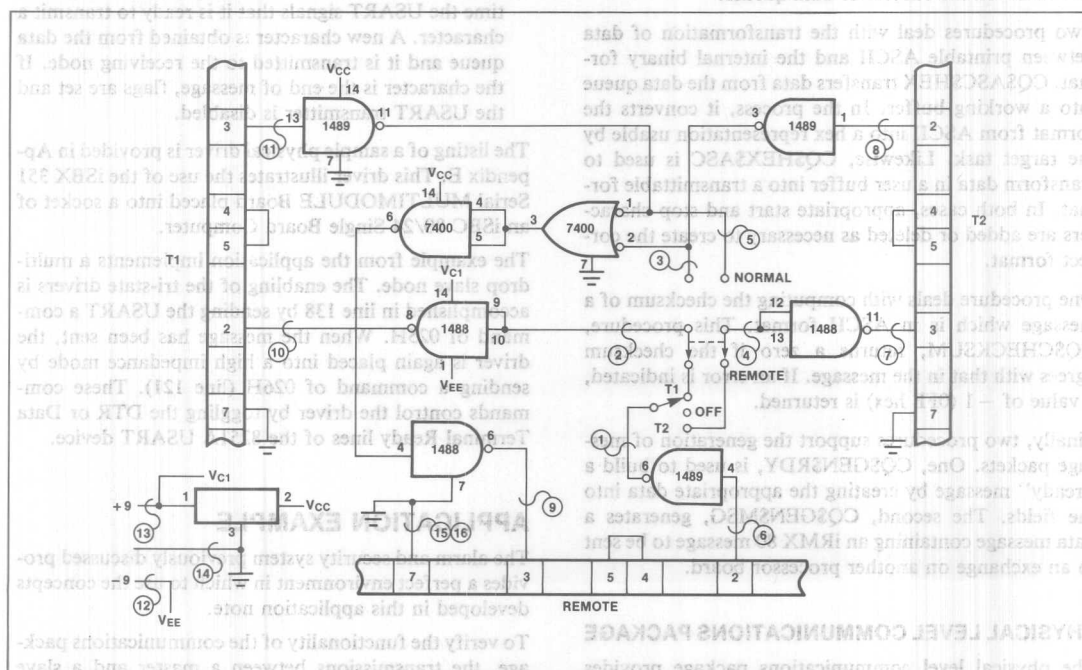


Figure 19. Communications Tee Schematic

The ability to break applications into small tasks, either functionally or geographically, aids the system designer by allowing his concentration on each task. A message transfer capability allows these tasks to again be tied together to form a complete solution to the application. Where the tasks reside on the same single board computer, the standard iRMX nucleus provides this ability. When different host boards are used, extensions to the nucleus allow the same functions to be performed. Both multiple processors on the same MULTIBUS chassis (refer to AP-88 and AP-112) and in different chassis using the serial links described in this application note can be supported with extensions to the nucleus.

Many concepts are used to create serial communication networks. Intel's single board computer products simplify the design process. Among these products, several stand out in this application. For example:

- **iRMX 80™ Executive** — This outstanding product simplifies the design of multitasking systems and provides a foundation for the implementation of extensions to create many varied configurations. Such things as task synchronization, interrupt handling, and message transfers allow multitasking. The free space manager allocates RAM and is an integral part of the serial communications link software. The terminal handler and the disk operating system simplify the software design by providing ready to use I/O drives which can be accessed by the user.
- **iSBC 80/10B™ Single Board Computer** — This microcomputer allows a low cost solution to many small to medium applications. The ability to operate the board in an iRMX 80 environment enhances its capabilities by allowing it to multiprocess. Its on-board serial communications link allows it to be used as a slave node in either EIA or current loop communications networks. The iSBX MULTIMODULE connector further enhances its capabilities by allowing customization of I/O to meet a wide variety of user applications.
- **iSBC 544™ Intelligent Communications Board** — The use of this board allows much of the protocol and physical drivers to be off-loaded from the host processor. The board provides an ideal master communications node for star type networks. By placing the handshaking operations on the communication board, the host is free to perform application-oriented functions with a much higher throughput rate.
- **iSBC 80/24™ Single Board Computer** — This powerful 8085A-2 based microcomputer board provides the capability to implement most of the system functions without the need to expand to additional memory or I/O expansion MULTIBUS boards. It fully supports the iRMX 80 executive. Its ability to act as a full bus master allows even greater flexibility through the implementation of MULTIBUS multiprocessor solutions. The two iSBX MULTIMODULE expansion connectors provide the user with considerable flexibility in the design of his system. The use of one of these sockets as a carrier for the serial communications MULTIMODULE board makes a multidrop serial communications link practical.
- **iSBX 351™ Serial MULTIMODULE Board** — The implementation of multidrop communications requires the use of an electrical interface which supports more than one communications node to share the link. The use of the RS422 operating mode of the board easily implements this feature. A well-defined hardware protocol also assists in the implementation of a serial multidrop communications link. Up to 11 slave nodes can be designed into a system using this powerful board. In addition, the iSBX 351 board can function in either the RS232C or the EIA mode for linking into terminals or for creating a point to point network.

The assistance of Judy McMillan in the implementation and testing of the security system and its communication links is greatly appreciated.

• **ISBC 544™ Intelligent Communications Board** — The use of this board allows much of the protocol and physical drivers to be off-loaded from the host processor. The board provides an ideal master communications node for star-type networks. By placing the handshaking operations on the communication board, the host is free to perform application-oriented functions with a much higher throughput rate.

• **ISBC 503A™ Single Board Computer** — This powerful 8085A-2 based microcomputer board provides the capability to implement most of the system functions without the need to expand to additional memory or I/O expansion MULTIBUS boards. It fully supports the iRMK 80 executive. Its ability to act as a full bus master also gives greater flexibility through the implementation of MULTIBUS master processor solutions. The iSBX MULTIMOD-ULE expansion connector provides the user with considerable flexibility in the design of his system. The use of one of these sockets as a carrier for the serial communications MULTIMODULE board makes a multistop serial communications link practical.

MULTIMODULE Board — The multistop communications module can interface which supports communications node to share RS-485 operating mode of the RS-485. A well-defined function is provided to assist in the implementation of slave nodes can be designed into a system using this powerful board. In addition, the iRMK 351 board can function in either the RS-232C or the EIA mode for linking into terminals or for creating a point to point network.

The assistance of Judy Mott in the implementation and testing of the security system and its communication links is greatly appreciated.

The ability to break applications into small tasks, either functionally or geographically, aids the system designer by allowing his concentration on each task. A message transfer capability allows these tasks to again be tied together to form a complete solution to the application. Where the tasks reside on the same single board computer, the standard iRMK nucleus provides this ability. When different board boards are used, extensions to the bus allow the same functions to be performed. Both multiple processors on the same MULTIBUS chassis (refer to AP-38 and AP-112) and in different chassis using the serial links described in this application note can be supported with extensions to the nucleus.

Main concepts are used to create serial communication networks. Intel's single board computer products simplify the design process. Among these products, several stand out for the application. For example:

• **iRMK 50T Executive** — This outstanding product simplifies the design of multitasking systems and provides a foundation for the implementation of extensions to create many varied configurations. Such as: task synchronization, interrupt handling, and message transfers allow multitasking. The free space manager allows RAM integral part of the serial communication system. The serial terminal handler and the disk of the software design by providing a way to use I/O drives which can be accessed.

• **ISBC 503A™ Single Board Computer** — This microcomputer allows a low-cost, high-performance small to medium applications. The ability to operate the board in an iRMK 80 environment enhances its capabilities by allowing it to multiprocess. Its on-board serial communications link allows it to be used as a slave node in either EIA or current loop communications network. The iSBX MULTIMODULE connector further enhances its capabilities by allowing customization of I/O to meet a wide variety of user applications.

APPENDIX A	2-189
APPENDIX B	2-193
APPENDIX C	2-205
APPENDIX D	2-215
APPENDIX E	2-236

```

1          $title('RMX/80 EXTENSION TO NAME EXCHANGES, VERSION 1.3')
          create$com$exch:      (exch$addr      exch$desc)
          do;
/* *****
CREATE$EXCHANGE creates a list of exchanges associating
the name of an exchange with its location, for the purpose
of other tasks wishing to find the location of an exchange.
FIND$EXCHANGE, when given the name of an exchange, polls
down the list and, when it finds a match, returns the
address to the requesting task.
*****
$no list
/* declare exchanges used by the task */
26 1      declare COM$CREATE$EXCH exchange$descriptor public;
27 1      declare CREATE$EXCH      exchange$descriptor public;
/* declare pointers and messages used by the task */
28 1      declare first$ptr address;
29 1      declare last$ptr  address;
30 1      declare msg$ptr   address;
31 1      declare exch$ptr  address;
32 1      declare m         byte;

33 1      declare request based msg$ptr structure(
          msg$hdr,
          exch$name(6) byte);

34 1      declare fs$req structure((
          msg$hdr,
          msg$length**address));
35 1      declare exch based exch$ptr structure(
          name(6)      byte,
          exchange(10) byte,
          link          address);

36 1      declare last$exch based last$ptr structure(
          name(6)      byte,
          exchange(10) byte,
          link          address);

37 1      declare response based msg$ptr structure (
          msg$hdr,
          exchange address );

38 1      NAMEX:
          Procedure(exptr) address public;
39 2          declare addr address;
40 2          declare exptr address;
41 2          declare (ex based exptr)(6) byte;

          /* declare the returning message with the address */

```

```

42 2 declare ret$msg based addr structure(msghdr,
    exch$adr address);

/* declare message to send to create a named exchange */
43 2 declare req structure(msg$hdr,
    name(6) byte);
/* create an exchange to communicate with
    the comm create exchange */
44 2 declare fsx exchangedescriptor;
45 2 call rcxch(.fsx);

/* build and send the request message */
46 2 req.length = 15;
47 2 call move(6, .ex, .req.name);
48 2 req.type = 200;
49 2 req.response$exchange = .fsx;
50 2 call rgsend(.comcreate$exch, req);
51 2 addr = rwait(.fsx, 0);
52 2 addr = ret$msg.exch$adr;

/* return with the address of the exchange */

53 2 return(addr);
54 2 end;

55 1 FIND$EXCH:
    procedure (name$ptr) address public;

    /*****
    This procedure finds the exchange having a name specified
    by the passed parameter pointer. If a match is found the
    exchange address is returned. If no match is found, a
    zero is returned.
    *****/

56 2 declare name$ptr address;
57 2 declare match byte;
58 2 declare (name based name$ptr)(6) byte;

/* test for no exchanges */
59 2 if first$ptr = 0
    then return 0;

/* test for a name match */
61 2 else do;
62 3     exch$ptr = first$ptr;
63 3     do while 1;
64 4         match = 0fffh;
65 4         do n = 0 to 5;
66 5             if name(n) <> exch.name(n)

```

```

68 5      then match = 0;
69 4      end;
71 4      if match then return .exch.exchange;
73 4      if exch.link = 0 then return 0;
74 4      exch$ptr = exch.link;
75 3      end;
76 2      return 0;
77 2      end;

78 1      CREATSCOM:
      procedure public;

      /*****
      This procedure creates exchange extensions when asked to
      do so by a task that wants its exchange made completely
      accessible. It saves the name of the exchange then
      creates an exchange to save its address. It may be updated
      at any time.
      *****/

      /* initialize exchanges */
79 2      call rqcxch(.com$create$exch);
80 2      call rqcxch(.create$exch);

      /* initialize pointers */
81 2      last$ptr = 0;
82 2      first$ptr = 0;

83 2      do while 1;

          /* get a request for creation of an exchange */
84 3      msg$ptr = rwait(.com$create$exch, 0);

          /* get some memory for the exchange */
85 3      fs$req.length = 11;
86 3      fs$req.type = 5;
87 3      fs$req.response$exchange = .create$exch;
88 3      fs$req.msg$length = 20;
89 3      call rsend(.rqfsax, .fs$req);
90 3      exch$ptr = rwait(.create$exch, 0);
91 3      exch$ptr = fs$req.msg$length;

          /* store name in the exchange structure */
92 3      call move( 5, .request.exch$name(0),
                  .exch.name(0));

          /* create an exchange */
93 3      call rqcxch(.exch.exchange(0));

          /* update the link field */

94 3      if last$ptr > 0
95 4      then do;
96 4          last$exch.link = exch$ptr;

```

```

97 4      last$ptr = exch$ptr;
98 4      exch.link = 0;
99 4      if match then return exch.exchange;
100 3      else do;
101 4          exch.link = 0;
102 4          first$ptr = exch$ptr;
103 4          last$ptr = exch$ptr;
104 4      end;

/* return the exchange pointer */
105 3      response.exchange = .exch.exchange;
106 3      call rqsend(request,response$exchange,msa$ptr);

107 3      end;
108 2      end;
109 1      end;
MODULE INFORMATION:
CODE AREA SIZE      = 01D2H      467D
VARIABLE AREA SIZE = 0040H      72D
MAXIMUM STACK SIZE = 0000H      4D
244 LINES READ
C PROGRAM ERROR(S)

/* initialize pointers */
last$ptr = 0;
first$ptr = 0;

do while 1;

/* get a request for creation of an exchange */
msa$ptr = rwait(.create$exch, 0);

/* get some memory for the exchange */
f$rec.length = 1;
f$rec.type = 5;
f$rec.response$exchange = .create$exch;
f$rec.msa$length = 0;
call r$send(.rf$exch, f$rec);
exch$ptr = rwait(.create$exch, 0);
exch$ptr = f$rec.msa$length;

/* store name in the exchange structure */
call move(0, request.exchange(0),
.exch.name(0));

/* create an exchange */
call r$exch(.exch.exchange(0));

/* update the link field */

if last$ptr > 0
then do;
last$exch.link = exch$ptr;

```



```

$title('Master Communications Protocol Driver, Version 8.1')
1  MASTER$DRIVER: do;
  /*
  The task contained in this listing supports the master
  communications protocol for establishing network commun-
  ications.
  The task must be configured using the ICU/80 as follows:
    TASK NAME: MASTER
    TASK ENTRY POINT: MLINK
    TASK PRIORITY: 112
    TASK STACK LENGTH: 100
    .
    .
    EXCHANGE: ROL7EX
    SCOPE: EXTERNAL
    INTERRUPT: YES
    EXCHANGE: TIMEX
    SCOPE: PUBLIC
    INTERRUPT: NO
    .
    LINK: :Fn:CQMSTR.OBJ
    .
    .
    LINK: :Fn:CQCOM.LIB
    LINK: :Fn:CQSLV.LIB
    Tasks desiring to send a message to a node should
    send a message to the exchange CQMIEX(node).
    Certain include files are used by the task.
    */
    $include (:ff:exch.elt)
2  1  =  DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
    =  MSG$HEAD ADDRESS,
    =  MSG$TAIL ADDRESS,
    =  TASK$HEAD ADDRESS,
    =  TASK$TAIL ADDRESS,
    =  NEXT$EXCH ADDRESS);
    $include (:ff:ied.elt)
3  1  =  DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
    =  MESSAGE$HEAD ADDRESS,
    =  MESSAGE$TAIL ADDRESS,
    =  TASK$HEAD ADDRESS,
    =  TASK$TAIL ADDRESS,
    =  EXCHANGE$LINK ADDRESS,
    =  LINK ADDRESS,
    =  LENGTH ADDRESS,
    =  TYPE BYTE);
    $include (:ff:msg.elt)
4  1  =  DECLARE MSG$HDR LITERALLY '

```

```

= LINK ADDRESS,
= LENGTH ADDRESS,
= TYPE BYTE,
= HOME$EXCHANGE ADDRESS,
= RESPONSE$EXCHANGE ADDRESS';
5 1 = DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
MSG$HDR,
= REMAINDER(1) BYTE)';
$include (:f0:common.elt)
6 1 = DECLARE TRUE LITERALLY 'OFFH';
7 1 = DECLARE FALSE LITERALLY '00H';
8 1 = DECLARE BOOLEAN LITERALLY 'BYTE';
9 1 = DECLARE FOREVER LITERALLY 'WHILE 1';
$include (:f0:synch.ext)
10 1 = RQSEND:
= PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
11 2 = DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
=
12 2 = END RQSEND;
=
13 1 = RQWAIT:
= PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
14 2 = DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
=
15 2 = END RQWAIT;
=
16 1 = RQACPT:
= PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
17 2 = DECLARE EXCHANGE$POINTER ADDRESS;
=
18 2 = END RQACPT;
=
19 1 = RQISND:
= PROCEDURE (IED$PTR) EXTERNAL;
20 2 = DECLARE IED$PTR ADDRESS;
=
21 2 = END RQISND;
$include (:f0:intrpt.ext)
22 1 = RQENDI:
= PROCEDURE EXTERNAL;
=
23 2 = END RQENDI;
=
24 1 = RQELVL:
= PROCEDURE (LEVEL) EXTERNAL;
25 2 = DECLARE LEVEL BYTE;
=
26 2 = END RQELVL;
=
27 1 = RQDLVL:
= PROCEDURE (LEVEL) EXTERNAL;
28 2 = DECLARE LEVEL BYTE;
=
29 2 = END RQDLVL;

```

```

30 1 = RQSETV:
    = PROCEDURE (PROC,LEVEL) EXTERNAL;
31 2 = DECLARE PROC ADDRESS;
32 2 = DECLARE LEVEL BYTE;
    =
33 2 = END RQSETV;
    =
34 1 = RQSETP:
    = PROCEDURE (PROC,LEVEL) EXTERNAL;
35 2 = DECLARE PROC ADDRESS;
36 2 = DECLARE LEVEL BYTE;
    =
37 2 = END RQSETP;
    $include (:ff:fsmgr.ext)
38 1 = DECLARE RQFSAX EXCHANGE$DESCRIPTOR EXTERNAL;
39 1 = DECLARE RQFSRX EXCHANGE$DESCRIPTOR EXTERNAL;
40 1 = DECLARE RQFREE EXCHANGE$DESCRIPTOR EXTERNAL;
    $include (:fl:masmsg.elt)
41 1 = declare msg$format literally 'structure (
    =   trgt   byte,
    =   cmd   byte,
    =   seq$la byte,
    =   seq$na byte,
    =   text(250) byte;
    =
42 1 = declare queue$format literally 'structure (
    =   end$ptr   byte,
    =   give$ptr  byte,
    =   take$ptr  byte,
    =   data$byte(254) byte;
    =
43 1 = declare par$format literally 'structure (
    =   na   byte,
    =   la   byte,
    =   run  byte,
    =   stop byte,
    =   que$pt byte;
    $include (:ff:objman.ext)
44 1 = RQCTSK:
    = PROCEDURE (STATIC$PTR) EXTERNAL;
45 2 = DECLARE STATIC$PTR ADDRESS;
    =
46 2 = END RQCTSK;
    =
47 1 = RQCXCH:
    = PROCEDURE (EXCHANGE$PTR) EXTERNAL;
48 2 = DECLARE EXCHANGE$PTR ADDRESS;
    =
49 2 = END RQCXCH;
    $include (:fl:comcom.ext)
50 1 = CQ$NEXT$GIVE:
    = PROCEDURE (q$ptr,give$byte) byte external;
51 2 = DECLARE q$ptr address;
52 2 = DECLARE give$byte byte;

```

```

53 2 = end cq$next$give;
54 1 = CQ$TAKE$TEST:
    = Procedure (q$ptr) byte external;
55 2 = Declare q$ptr address;
56 2 = end cq$take$test;
57 1 = CQ$NEXT$TAKE:
    = Procedure (q$ptr) byte external;
58 2 = Declare q$ptr address;
59 2 = end cq$next$take;
60 1 = CQ$Q$INIT:
    = Procedure (q$ptr,q$size) external;
61 2 = Declare q$ptr address;
62 2 = Declare q$size byte;
63 2 = end cq$q$init;
64 1 = CQ$ASC$HEX:
    = Procedure (q$ptr,msg$ptr) external;
65 2 = Declare (q$ptr,msg$ptr) address;
66 2 = end cq$asc$hex;
67 1 = CQ$CHECKSUM:
    = Procedure (q$ptr) byte external;
68 2 = Declare q$ptr address;
69 2 = end cq$checksum;
70 1 = CQ$HEX$ASC:
    = Procedure (q$ptr,hex$byte) external;
71 2 = Declare q$ptr address;
72 2 = Declare hex$byte byte;
73 2 = end cq$hex$asc;
74 1 = CQ$MSG$OUT:
    = Procedure (msg$size,q$ptr,par$ptr,msg$ptr) external;
75 2 = Declare msg$size byte;
76 2 = Declare (q$ptr,par$ptr,msg$ptr) address;
77 2 = end cq$msg$out;
78 1 = CQ$GEN$RDY:
    = Procedure (trget,msg$ptr,q$ptr,par$ptr) external;
79 2 = Declare trget byte;
80 2 = Declare (msg$ptr,q$ptr,par$ptr) address;
81 2 = end cq$gen$rdy;
82 1 = CQ$GEN$MSG:
    = Procedure (msg$pointer,trget,msg$ptr,q$ptr,
        par$ptr,save$flg) external;
83 2 = Declare (trget,save$flg) byte;
84 2 = Declare (msg$pointer,msg$ptr,q$ptr,par$ptr) address;
85 2 = end cq$gen$msg;

86 1  USRINT:
    Procedure external;
87 2  end usrint;
88 1  FIND$EXCH:
    Procedure (name$ptr) address external;
89 2  declare name$ptr address;
90 2  end find$exch;
91 1  NAMEX:
    Procedure (exptr) address external;
92 2  declare exptr address;
93 2  end

```

```

94 1      return$ram:
          procedure(pointer) external;
95 2      declare pointer address;
96 2      end return$ram;
          /* a pointer is used to store the address of exchanges
          /*
          Certain literals are used to define the network's
          physical characteristics. These are:
          */
97 1      Declare N$NODE literally '0'; /* number of nodes */
98 1      Declare NODE$ARRAY literally '5';
          /*
          A structure provides the data queues for the
          transmission of data to each node. It is defined
          and is available as a public
          */
99 1      Declare COMSTQ queue$format public at (811ch);
          /*
          A single data queue is used to support the input of
          data from a node since only one slave is given a
          window at a time:
          */
100 1      Declare CQMSIQ queue$format public at (801bh);
          /*
          Each node has an associated set of flags which
          indicate the operational mode of that node. The
          function of each bit is defined as:
          bit 0 - request synchronization (synch$request)
          bit 1 - request initialization (init$request)
          bit 2 - repeat last message (repeat$request)
          bit 3 -
          bit 4 -
          bit 5 -
          bit 6 -
          bit 7 - error flag (error$flag)
          */
101 1      Declare CQCMDF(node$array) byte public;
102 1      Declare SYNCH$REQUEST literally 'C1H';
103 1      Declare INIT$REQUEST literally 'C2H';
104 1      Declare REPEAT$REQUEST literally 'C4H';
105 1      Declare ERROR$FLAG literally 'CCH';
          *****
          /*
          A counter is used to indicate the number of attempts
          to establish communications with a node. When it
          reaches a maximum value, a synch will be sent to the
          node. when the maximum value of synchs are sent to a
          node with no effect, a reset will be sent to the node.
          */
          *****

```



```

106 1      Declare ERROR$COUNT(node$array) byte;
107 1      Declare MAX$COUNT literally '5';
108 1      Declare SYNCH$COUNT(node$array) byte;

/* a pointer is used to store the address of exchanges
   used by incoming messages */

109 1      Declare target$exch address;
/*
   A set of exchanges is used to hold a message until it
   has been acknowledged by the slave.
*/
110 1      Declare HOLD$EXCH(node$array) exchange$descriptor;
/*
   A set of exchanges is provided to accept messages which
   are to be sent to any of the nodes.
*/
111 1      Declare CQMIEX(node$array) exchange$descriptor public;
/*
   The task uses various sets of data structures
*/
112 1      Declare req$msg structure (
           msg$hdr,
           msg$length address);
113 1      Declare (m,n,node$cnt,outmode,ram$size) byte;
114 1      Declare msg$ptr address;
115 1      Declare msg msg$format;
116 1      declare start$544 byte public at (8000h);
117 1      Declare data$block based msg$ptr structure (
           msg$hdr);
118 1      Declare CQMPAR(node$array) par$format public
           at (8002h);
119 1      Declare RAM$MSG based msg$ptr structure (
           msg$hdr);
120 1      Declare CQ$ACTIVE$NODE byte public at (8001h);
121 1      Declare free$mem address;
122 1      Declare free$msg based free$mem structure(msg$hdr);
/*
   The task uses certain exchanges for internal
   communications
*/
123 1      Declare CQMSEX exchange$descriptor;
124 1      Declare RQL7EX int$exchange$descriptor public;

Seject
/*****
           error$test
This short procedure is used to increment the error
counters and to signal an initialization or synch command
when required. It will order a re-transmission of the
last message each time an error is detected.
*****/

```

```

125 1      error$test: procedure;
126 2          If (error$count(n):=error$count(n)+1) > max$count
127 3              then do;
128 3                  error$count(n)=0;
129 3                  if (synch$count(m):=synch$count(m)+1) > max$count
130 4                      then do;
131 4                          synch$count(m) = 0;
132 4                          cqcndf(n) = cqcndf(n) or error$flag
133 4                              or init$request;
134 3                      end;
135 3                  else cqcndf(n) = cqcndf(n)
136 3                      or error$flag or synch$request;
137 2                  end;
138 2                  else cqcndf(n) = cqcndf(n) or repeat$request;
139 2                  return;
140 2          error$test;
141 2          Seject
142 1      MLINK: Procedure public;

143 2          /* Initialize the system and the task */
144 3          Do n=0 to NSNODE;
145 3              CQCNDf(n)=synch$request or error$flag;
146 3              CQMPAR(n).run,CQMPAR(n).stop,CQMPAR(n).que$pt=0;
147 3              ERROR$COUNT(n)=0;
148 3              SYNCH$COUNT(n)=0;
149 3          end;

150 2          /* Initialize the node pointer */
151 2          NODE$CNT=0;

152 2          /* Initialize the input exchanges */
153 3          Do N=0 to n$node;
154 3              call RQCXCH(.CQMIEX(n));
155 3              call RQCXCH(.HCLD$EXCH(n));
156 3          end;

157 2          call RQCXCH(.COMSEX);

158 2          /* Initialize the queues */
159 2          call cq$qs$init(.cqmstq, 254);
160 2          call cq$qs$init(.cqmsiq, 254);

161 2          /* Create the named exchanges and give ram to fsm*/
162 2          call usrint;

163 2          /* initialize the level 7 interrupt exchange */
164 2          call rqlvl(7);

165 2          /* Begin main task loop */
166 2          Do forever;

167 2          /* Begin support of one nodal channel */
168 3          Do n=1 to n$node;

```

```

158 4      /* reset queue pointers */
159 4      camstq.give$ptr, camstq.take$ptr = 0;

160 4      /* Compute nodal mode number */
161 4      If (cacmdf(n) and synch$request) > 0
162 4      then outmode = 1;
163 4      else outmode = 0;
164 4      If (cqcmdf(n) and init$request) > 0
165 4      then outmode = 2;
166 4      cq$active$node = n;

167 4      /* if comm failure, kill all messages */
168 4      if (cqcmdf(n) and error$flag) > 0
169 4      then do;
170 5          do while (msg$ptr:=rqacpt(.comex(n))) > 0;
171 6              call return$ram(msg$ptr);
172 6          end;
173 5          do while (msg$ptr:=rqacpt(.hold$exch(n))) > 0;
174 6              call return$ram(msg$ptr);
175 6          end;
176 5      end;

177 4      /* Operate on nodal mode */
178 4      Do case outmode;

179 5          /* case 0, routine communications */
180 5          Do;
181 6              If (cqcmdf(n) and repeat$request) > 0
182 6              then do;
183 7                  /* support of retransmission request */
184 7                  cqcmdf(n)=cqcmdf(n)
185 7                  and not repeat$request;
186 7                  msg$ptr=rqacpt(.hold$exch(n));
187 7                  if msg$ptr>0
188 7                  then do;
189 8                      /* retransmit old message */
190 8                      then do;
191 9                          if campar(n).na = 0
192 9                          then campar(n).na = 15;
193 9                          else campar(n).na=campar(n).na
194 9                          - 1;
195 9                          call cq$gen$msg(msg$ptr,n,.msg,
196 9                          .camstq,.cqpar(n),0);
197 9                          call rqsend(.hold$exch(n),
198 9                          msg$ptr);
199 9                          if campar(n).na = 15
200 9                          then campar(n).na = 0;
201 9                          else campar(n).na
202 9                          =campar(n).na+1;
203 8                      end;
204 8              end;

```

```

191 7      /* no old message, send ready */
192 8      else do;
193 8          msg.trgt=n;
194 8          msg.cmd=3;
195 8          call cq$msg$out(0,.cqmsta,
196 8              .cqmpar(n),.msg);
197 8      end;
198 7      /* end of retransmission */
199 7      end;
200 6      /* support of next message */
201 7      else do;
202 7          /* clear hold exchange */
203 7          msg$ptr=rqacpt(.hold$exch(n));
204 7          if msg$ptr>0
205 7          then do;
206 7              /* free space return size */
207 7              ramsize=ata$block.length;
208 7              if (ramsize mod 4) > 0
209 7              then ramsize=ata$block.length
210 7              + (4-(ramsize mod 4));
211 7              call RQSEND(.rqfsrx,msg$ptr);
212 7          end;
213 7          /* test for a message output */
214 7          msg$ptr=rqacpt(.cqmex(n));
215 7          /* when a message exists */
216 7          if msg$ptr>0
217 7          then do;
218 7              if (target$exch:
219 7                  =find$exch(msg$ptr+9))> 0
220 7              then call
221 7                  rqsend(target$exch,msg$ptr);
222 7              else do;
223 7                  call cq$gen$msg(msg$ptr,n,.msg,
224 7                      .cqmsta,.cqmpar(n),0);
225 7                  call rqsend(.hold$exch(n),
226 7                      msg$ptr);
227 7                  /* increase the na flag */
228 7                  if cqmpar(n).na = 15
229 7                  then cqmpar(n).na = 0;
230 7                  else cqmpar(n).na
231 7                      = cqmpar(n).na + 1;
232 7              end;
233 7          end;
234 7          /* when no message exists */
235 7          else do;
236 7              msg.trgt=n;
237 7              msg.cmd=3;

```

```

222 \* 8/when msg is ready, send msg \* call cq$msg$out(f,.cqmsta,
                                     .cqmpar(n),.msg);
223 8      ;n:=p1.pam      end;
224 7      ;msg:=p2.pam    end; /* end of routine message */
                                     call cq$msg$out(f,.cqmsta,
                                     .cqmpar(n),.msg); /* start the message transmission */
225 6      ;start$544 = n;
/* end of transmission */
226 6      end; /* end of case 1 */
/* support of next message */
/* case 1, synch request */
227 5      do;
/* clear hold exchange */
228 6      ;cqmpar(n).na,cqmpar(n).la = 0;
229 6      ;msg.trgt=n;
230 6      ;msg.cmd=1;
231 6      call cq$msg$out
                                     (f,.cqmsta,.cqmpar(n),.msg);
232 6      ;start$544 = n;
233 6      ;cqcmdf(n)=cqcmdf(n)
                                     and not synch$request;
234 6      ;end;
                                     call ROEND(.rdrtr,msgptr);
/* case 2, initialize request */
235 5      do;
/* test for a message output */
236 6      ;cqcmdf(n)=(cqcmdf(n) and
                                     not init$request) or synch$request;
237 6      ;cqmpar(n).na, cqmpar(n).la = 0;
238 6      ;msg.trgt=n;
239 6      ;msg.cmd=0;
240 6      ;call cq$msg$out
                                     (f,.cqmsta,.cqmpar(n),.msg);
241 6      ;start$544 = n;
242 6      ;end;
                                     call ROEND(.rdrtr,msgptr);
243 5      ;end; /* end of do case blocks */
/* wait for a response from the slave */
244 4      ;msg$ptr = rwait(.rql7ex,25);
245 4      ;start$544 = 0;
/* poll on the slave */
246 4      ;if (cqcmdf(n) or (ram$msg.type=3
                                     and cqcmdf(n).na = 0))
                                     then
/* test for max number of errors to node */
247 4      ;call nerror$test;
/* support of good response return */
248 4      ;else
                                     ;n:=p1.pam
                                     ;p:=p2.pam
                                     /* test for good checksum */

```



```

249 5 = an.la$seq$mpo, if cdc$checksum(.cqm$seq)=0
      then do;
      /* test for a good slave LA response */
      L + mpo$seq.para = mpo$seq.para + 1 /* convert message to hex format */
251 6 call bcq$asc$hex(.cqm$seq,.msg);
      else do;
      an.la$seq$mpo <> mpo$seq$mpo /* test for data message receipt */
252 6 if mpo$cmd = 2
      or mpo$seq$mpo = mpo$seq$mpo
      /* support receipt of data message */
      then do;
      cqm$seq$mpo = cqm$seq$mpo + 1
      cqm$seq$mpo = cqm$seq$mpo + 1
      /* get ram from free space manager */
254 7 req$msg.length=11;
255 7 req$msg.type = 5;
256 7 req$msg.response$exch=cdc$seq$ex;
257 7 req$msg.msg$length=msg.text(2);
258 7 call rcsend(.rqf$ex,.req$msg);
259 7 msg$ptr=rwait(.cdc$seq$ex,0);
260 7 msg$ptr=req$msg.msg$length;
      /* end of message arrived options */
      /* move message into memory */
261 7 call move(msg.text(2),msg.text(2)
      ,msg$ptr);

      /* if target is another node.. */
      if msg.trgt > 0
      then call rcsend(.cqm$ex(msg.trgt)
      ,msg$ptr);

      /* if target is master board */
264 7 if (target$exch:= find$exch(msg$ptr +
9))
      > 0
      then
265 7 call rcsend(target$exch, msg$ptr);
266 7 else do;
267 8 rsize = msg.text(2);
268 8 if (rsize mod 4) > 0
      then msg.text(2)=msg.text(2)
      + (4 - (rsize mod 4));
270 8 call rcsend(.rqf$rx, msg$ptr);
271 8 end;

      /* increment message counter */
272 7 if cqm$par(n).la = 15
      then cqm$par(n).la = 0;
274 7 else cqm$par(n).la=cqm$par(n).la+1;

275 7 end;
      /* respond to non-sequence acknowledge */
276 6 if msg.cmd = 5

```

```

n=msg; then cqm$par(n).la,cqm$par(n).na = 0;
/* test for a good slave LA response */
278 6 /* if cqm$par(n).na = msg.seqla + 1
    if cqm$par(n).na = msg.seqla + 1
    then call error$test;
280 6 else do;
281 7 /* if msg.seqla <> cqm$par(n).na
    then cqm$df(n) = cqm$df(n)
    or synchrequest;
283 7 else do;
284 8 error$count(n) = 0;
285 8 cqm$df(n) = cqm$df(n)
    and not error$flag;
/* get cm from free space message */
286 8 end;end;
287 7 end;end;
288 6 end; /* end of response return */
/* support of bad checksum */
289 5 else call error$test;
290 5 end;end;
/* end of message arrived options */
291 4 end;end;
292 3 /* move message to memory */
293 2 call move(msg.text,msg.text)
294 1 end;end;
/* if target is another node */
if msg.fid > 0
then call readn(context(msg.fid),
msg$ptr);
/* if target is master board */
if (target$exch = find$exch(msg$ptr) +
1)
then
call readn(target$exch,msg$ptr);
else do;
resize = msg.text(2);
if (resize mod 4) > 0
then msg.text(2) = msg.text(2)
+ 4 - (resize mod 4);
call readn(target$exch,msg$ptr);
end;
/* increment message counter */
if cqm$par(n).la = 0
then cqm$par(n).la = 1;
else cqm$par(n).la = cqm$par(n).la + 1;
end;
/* respond to non-secure acknowledge */
if msg.cmd = 5

```

```

1      $title('Slave Communications Package Version 3.2')
      COMSLV$MODULE: do;
/*
      This is the slave communication main task.
      It should be configured into the system
      having the following parameters:

      Task name- CQSLAV
      Task entry point- CQSLAV
      Task priority- as required
      Task stack length- 100

      The LINK and LOCATE commands of the user
      must include the following statements:

      Link ...
      ...
      :Fn:CQSLAV.OBJ
      :Fn:CQS351.OBJ
      :Fn:CQCOM.LJP
      :Fn:CQSLV.LJP
      ...

      Several system parameters must be defined
      in the user code to define the mode
      characteristics. These parameters are defined
      as:
      co$slad- a byte value providing the
              nodal address of the commun-
              ication node.
      fre$mem- an address value which provides
              a pointer to a block of RAM to
              be assigned via the free space
              manager to the communications.
      ram$len- an address value which indicates
              the size of the above block.

      */
2      1      declare co$slad byte external;
      $nolist
48      1      $include(:fl:commsg.elt)
      =      declare msg$format literally 'structure (
      =          trgt      byte,
      =          cmdnd      byte,
      =          seq$la      byte,
      =          seq$na      byte,
      =          text(250) byte )';
      =
49      1      =      declare queue$format literally 'structure (
      =          end$ptr      byte,
      =          give$ptr      byte,
      =          take$ptr      byte,

```

```

data$byte(254) byte)';
50 1 = declare par$format literally 'structure (
    =     la     byte,
    =     na     byte,
    =     run     byte,
    =     stop    byte,
    =     que$pt  byte)';
    $include(:ff:objman.ext)
51 1 = RQCTSK:
    =     PROCEDURE (STATIC$PTR) EXTERNAL;
52 2 =     DECLARE STATIC$PTR ADDRESS;
    =
53 2 =     END RQCTSK;
    =
54 1 = RQCXCH:
    =     PROCEDURE (EXCHANGES$PTR) EXTERNAL;
55 2 =     DECLARE EXCHANGES$PTR ADDRESS;
    =
56 2 =     END RQCXCH;
    $include(:fl:comcom.ext)
57 1 = CQ$NEXT$GIVE:
    =     Procedure (q$ptr,give$byte) byte external;
58 2 =     Declare q$ptr address;
59 2 =     Declare give$byte byte;
60 2 =     end cq$next$give;
61 1 = CQ$TAKE$TEST:
    =     Procedure (q$ptr) byte external;
62 2 =     Declare q$ptr address;
63 2 =     end cq$take$test;
64 1 = CQ$NEXT$TAKE:
    =     Procedure (q$ptr) byte external;
65 2 =     Declare q$ptr address;
66 2 =     end cq$next$take;
67 1 = CQ$Q$INIT:
    =     Procedure (q$ptr,q$size) external;
68 2 =     Declare q$ptr address;
69 2 =     Declare q$size byte;
70 2 =     end cq$q$init;
71 1 = CQ$ASC$HEX:
    =     Procedure (q$ptr,msg$ptr) external;
72 2 =     Declare (q$ptr,msg$ptr) address;
73 2 =     end cq$asc$hex;
74 1 = CQ$CHECKSUM:
    =     Procedure (q$ptr) byte external;
75 2 =     Declare q$ptr address;
76 2 =     end cq$checksum;
77 1 = CQ$HEX$ASC:
    =     Procedure (q$ptr,hex$byte) external;
78 2 =     Declare q$ptr address;
79 2 =     Declare hex$byte byte;
80 2 =     end cq$hex$asc;
81 1 = CQ$MSG$OUT:
    =     Procedure (msg$size,q$ptr,par$ptr,msg$ptr) external;
82 2 =     Declare msg$size byte;

```

```

83 2 = Declare (q$ptr,par$ptr,msg$ptr) address;
84 2 = end cq$msg$out;
85 1 = CQ$GEN$RDY:
86 2 = Procedure (trget,msg$ptr,q$ptr,par$ptr) external;
87 2 = Declare trget byte;
88 2 = Declare (msg$ptr,q$ptr,par$ptr) address;
89 1 = CQ$GEN$MSG:
      = Procedure (msg$pointer,trget,msg$ptr,q$ptr,par$ptr,sav
      - e$flg) external;
90 2 = Declare (trget,save$flg) byte;
91 2 = Declare (msg$pointer,msg$ptr,q$ptr,par$ptr) address;
92 2 = end cq$gen$msg;
93 1 Declare RQL6EX int$exchange$descriptor public;
94 1 Declare CQ$IEX exchange$descriptor public;
95 1 Declare cmrgex exchange$descriptor;
96 1 declare mld$slave$ex exchange$descriptor public;
97 1 Declare cotime exchange$descriptor public;
98 1 cq$start$msg$351:
      procedure external;
99 2 end cq$start$msg$351;
100 1 cq$init$351:
      procedure external;
101 2 end cq$init$351;
102 1 find$exch:
      procedure(name$ptr) address external;
103 2 declare name$ptr address;
104 2 end find$exch;
105 1 cq$startup:
      procedure external;
106 2 end cq$startup;
107 1 declare cq$in$sl queue$format public;
108 1 declare cq$out$sl queue$format public;
109 1 declare msg msg$format;
110 1 declare cq$par$ par$format public;
111 1 declare fre$mem address;
112 1 declare cqcmdf(5) byte external;
113 1 declare fre$msg based fre$mem structure (
      msg$hdr );

```

/*

The message which is transmitted between nodes follows the description below:

The complete message structure is defined as:

STX TARGET COMMAND SEQ TEXT CHECKSUM EOT


```
-----
I LF I CC-FF I C-F I LA,NA I n I CC-FF I CR I
-----
```

```
0 1 2 3 4 5 6 7+n 8+n
```

With the exception of the start of text (a line feed) and the end of transmission (a carriage return), all characters transmitted in the message string will consist of printable ASCII characters.

The target field consists of two frames which represent the hex address of the device to which the message is addressed. The protocol allows for up to 256 unique device addresses.

The command field indicates the type of message which is being sent in the network. It consists of a single frame representing a hex number in the range from 0 to FFH. The current message definitions are:

command	label
0	INIT
1	SYNCH
2	DATA
3	READY
4	NOT READY
5	NON SEQ ACK
6	reserved
.	"
.	"
.	"
F	"

This is the main link level RMX/RC communications task which supports slave operations of a single board computer. Several high level functions are supported by this task.

Messages containing data may be sent or received by this task. To send a message to another processor on the communications loop, a message of the format shown is placed into the communications exchange, CQSIEX. When the message has been transmitted, the RAM area in which the message was located will be returned to the free space manager.

Messages may also be received by the board when they are addressed to the communications address assigned to this board. When a message has been received, it will be transferred to a block of RAM obtained from the free space manager and then sent to an exchange which is designated in the name field of the message.


```

138 6      name = .msg.text + 9;
139 6      cqcndf(0) = cqcndf(0) and 7fh;

/* handle each message type by case */
140 6      do case msg.cmd;
/* case 0, INIT command */
141 7      call cqstartup;

/* case 1, SYNCH command */
142 7      do;
/* reset counters */
143 8      cq$pars.la, cq$pars.na = 0;
/* initialize data queues */
144 8      call cq$init (.cq$in$sl, 250);
145 8      call cq$init (.cq$out$sl, 250);
/* return non seq ack message */
146 8      msg.trgt = 0;
147 8      msg.cmd = 5;
148 8      call cq$msg$out (0, .cq$out$sl, cq$
- pars, .msg);
149 8      call cq$start$msg$sl;
150 8      /* initialize task at power up */
/* call cq$init (.cq$in$sl, 250);
/* call cq$init (.cq$out$sl, 250);
/* case 2, DATA command */
151 7      cq$pars.run, cq$pars.stop, cq$pars.duept = 0;

/* test for correct NA */
152 8      if cq$pars.na = msg.seq$na
/* clear old messages */
154 9      msg$ptr = msg$ptr + 1;
- x);
155 9      if msg$ptr > 0
then call return$ram(msg$ptr);
/* wait for a window from the master */
/* increment LA counter */
157 9      if (cq$pars.la = cq$pars.la+1)
/* test for good communications with master */
then cq$pars.LA = 0;
/* verify that exchange exists */
/* test for the receipt of a valid message */
159 9      if cq$checksum(.cq$in$sl) = 0
160 9      TARGET$EXCH = FIND$EXCH(na);
if TARGET$exch > 0
then do;
/* get RAM and store message */
162 10      req$msg.length = 11;
163 10      req$msg.type = 5;

```

```

164 10      = .cm$rq$ex;
165 10      ext(2);
166 10      msg);
167 10      x, 0);
168 10      th;
169 10      g.text(0), msg$ptr);
170 10      msg$ptr);
171 10
172 9      msg$ptr = RQACPT (.cq$ex);
173 9      if msg$ptr = 0
174 9      then call cq$gen$rdy (0, .msg, .
175 9      - cq$out$sl, .cq$pars);
176 10      if (target$exch = find$exc
177 10      - h(msg$ptr + 9)) > 0
178 10      - ch, msg$ptr);
179 11      call cq$gen$msq (msg$ptr,
180 11      - 0, .msg, .cq$out$sl, .cq$pars, 0);
181 11      - e$ex, msg$ptr);
182 10      end;
183 9      /* if bad na, then retransmit last
184 8      msg */
185 9      if msg$ptr > 0
186 9      then do;
187 9      call cq$gen$msq (msg$ptr, 0,
188 10      - .msg, .cq$out$sl, .cq$pars, 0);
189 10      - , msg$ptr);
190 10
191 10      end;
192 9      /* static transmission */

```

```

req$msg.response$exch
req$msg.msg$length = msg.t
call RQSEND (.rqf$ex, .req$
msg$ptr = RQWAIT (.cm$rq$e
msg$ptr = req$msg.msg$len
call move (msg.text(2), .ms
call RQSEND (target$exch,
end;
/* test for data out request */
msg$ptr = RQACPT (.cq$ex);
if msg$ptr = 0
then call cq$gen$rdy (0, .msg, .
else do;
if (target$exch = find$exc
then call rqsend(target$ex
else do;
call cq$gen$msq (msg$ptr,
call rqsend(.hold$slave
end;
end;
end;
/* if bad na, then retransmit last
else do;
cq$pars.na = msg.seq$na;
msg$ptr = rqacpt(.hold$slave$e
if msg$ptr > 0
then do;
call cq$gen$msq (msg$ptr, 0,
call rqsend(.hold$slave$ex
end;
end;
end;

```

```

193 8      /* send message */
194 8      call cq$start$msg$351;
      end;

195 7      /* case 3, RDY command */
      do;

196 8          /* verify na is correct */
          if cq$pars.na = msg.seq$na
          then do;

197 9              /* clear old messages */
              msg$ptr = rdacpt(.hold$slave$e
              if msg$ptr > 0
              then call return$ram(msg$ptr);

              /* test for a message waiting
              msg$ptr = RQACPT(.cq$ix);
              if msg$ptr = 0
              then call cq$gen$rdy(0, msg, .
              else do;
                  if (target$exch:=find$exch
                  then call rdaend(target$ex
                  else do;
                      call cq$gen$msg(msg$ptr
                      call rdaend(.hold$slave$ex
                      end;
                  end;
              end;

202 9      end;

203 9      /* if bad na, then retransmit last
      */
      else do;
          cq$pars.na = msg.seq$na;
          msg$ptr = rdacpt(.hold$slave$e
          if msg$ptr > 0
          then do;
              call cq$gen$msg(msg$ptr, 0,
              call rdaend(.hold$slave$ex
              end;
          end;

204 9      /* start transmission */
205 10      end;

206 10      /* if bad na, then retransmit last
      */
      else do;
          cq$pars.na = msg.seq$na;
          msg$ptr = rdacpt(.hold$slave$e
          if msg$ptr > 0
          then do;
              call cq$gen$msg(msg$ptr, 0,
              call rdaend(.hold$slave$ex
              end;
          end;

207 10      /* start transmission */
208 10      end;

209 10      /* if bad na, then retransmit last
      */
      else do;
          cq$pars.na = msg.seq$na;
          msg$ptr = rdacpt(.hold$slave$e
          if msg$ptr > 0
          then do;
              call cq$gen$msg(msg$ptr, 0,
              call rdaend(.hold$slave$ex
              end;
          end;

210 10      /* start transmission */
211 10      end;

212 9      /* if bad na, then retransmit last
      */
      else do;
          cq$pars.na = msg.seq$na;
          msg$ptr = rdacpt(.hold$slave$e
          if msg$ptr > 0
          then do;
              call cq$gen$msg(msg$ptr, 0,
              call rdaend(.hold$slave$ex
              end;
          end;

213 8      /* start transmission */
214 8      end;

215 8      /* if bad na, then retransmit last
      */
      else do;
          cq$pars.na = msg.seq$na;
          msg$ptr = rdacpt(.hold$slave$e
          if msg$ptr > 0
          then do;
              call cq$gen$msg(msg$ptr, 0,
              call rdaend(.hold$slave$ex
              end;
          end;

216 9      /* start transmission */
217 9      end;

218 9      /* if bad na, then retransmit last
      */
      else do;
          cq$pars.na = msg.seq$na;
          msg$ptr = rdacpt(.hold$slave$e
          if msg$ptr > 0
          then do;
              call cq$gen$msg(msg$ptr, 0,
              call rdaend(.hold$slave$ex
              end;
          end;

219 10      /* start transmission */
220 10      end;

221 9      /* if bad na, then retransmit last
      */
      else do;
          cq$pars.na = msg.seq$na;
          msg$ptr = rdacpt(.hold$slave$e
          if msg$ptr > 0
          then do;
              call cq$gen$msg(msg$ptr, 0,
              call rdaend(.hold$slave$ex
              end;
          end;

```



```

222 8      call cq$start$msg$351;
223 8      end;

/* case 4, NONRDY command */
224 7      do;end;

/* case 5, NONSEQACK command */
226 7      do;end;

/* case 6, reserved */
228 7      do;end;

/* case 7, reserved */
230 7      do;end;

/* case 8, reserved */
232 7      do;end;

/* case 9, reserved */
234 7      do;end;

/* case A, reserved */
236 7      do;end;

/* case B, reserved */
238 7      do;end;

/* case C, reserved */
240 7      do;end;

/* case D, reserved */
242 7      do;end;

/* case E, reserved */
244 7      do;end;

/* case F, reserved */
246 7      do;end;

248 7      end; /* do case */
249 6      end; /* good target */
250 5      end; /* good checksum */
251 4      end; /*good communications with iMOS*/

/* clear out messages if com failure exists */
252 3      else do;
253 4          do while (msg$ptr:=rqacpt(.cdsiex)) > 0;
254 5              call return$ram(msg$ptr);
255 5          end;

256 4          do while (msg$ptr:=rqacpt(.hold$slave$ex)) > 0;
257 5              call return$ram(msg$ptr);
258 5          end;

/* set failure indicator */

```

```

2599 4 1 cqcndf(F) = cqcndf(E) or 8fh;
260 4 end;

261 3 end; /* do forever */
262 2 end ccslav;
263 1 end comslv$module;

/* case 5, NONEEDACK command */
do;end;

/* case 6, reserved */
do;end;

/* case 7, reserved */
do;end;

/* case 8, reserved */
do;end;

/* case 9, reserved */
do;end;

/* case A, reserved */
do;end;

/* case B, reserved */
do;end;

/* case C, reserved */
do;end;

/* case D, reserved */
do;end;

/* case E, reserved */
do;end;

/* case F, reserved */
do;end;

end; /* do case */
end; /* good target */
end; /* good checksum */
end; /* good communications with host */

/* clear out messages if com failure exists */
else do;
do while (msg$ptr=rcv$ptr(.coslex)) > 0;
call return$rm(msg$ptr);
end;

do while (msg$ptr=rcv$ptr(.hold$slave$ex)) > 0;
call return$rm(msg$ptr);
end;

/* set failure indicator */

```

```

1      $title('QUEUE INITIALIZATION PROCEDURE')
2      $INIT$MODULE: Do;
3      1 = declare eom literally 'CDH';
4      1 = $include(:fl:commsg.elt)
5      1 = declare msg$format literally 'structure (
6      =     trgt      byte,
7      =     cmdn      byte,
8      =     seq$la     byte,
9      =     seq$na     byte,
10     =     text(25C) byte )';
11     1 = declare queue$format literally 'structure (
12     =     end$ptr    byte,
13     =     give$ptr   byte,
14     =     take$ptr   byte,
15     =     data$byte(254) byte )';
16     1 = declare pars$format literally 'structure (
17     =     la         byte,
18     =     na         byte,
19     =     run        byte,
20     =     stop       byte,
21     =     que$pt     byte )';
22     1 = $include(:f0:synch.ext)
23     1 = RQSEND:
24     =     PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
25     2 = DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
26     =
27     2 = END RQSEND;
28     =
29     1 = RQWAIT:
30     =     PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
31     2 = DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
32     =
33     2 = END RQWAIT;
34     =
35     1 = RQACPT:
36     =     PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
37     2 = DECLARE EXCHANGE$POINTER ADDRESS;
38     =
39     2 = END RQACPT;
40     =
41     1 = RQISND:
42     =     PROCEDURE (IED$PTR) EXTERNAL;
43     2 = DECLARE IED$PTR ADDRESS;
44     =
45     2 = END RQISND;
46     1 = $include(:f0:exch.elt)
47     1 = DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
48     =     MSG$HEAD ADDRESS,
49     =     MSG$TAIL ADDRESS,
50     =     TASK$HEAD ADDRESS,
51     =     TASK$TAIL ADDRESS,

```

```

= NEXT$EXCH ADDRESS);
$include(:f0:msg.elt)
19 1 = DECLARE MSG$HDR LITERALLY '
= LINK ADDRESS,
= LENGTH ADDRESS,
= TYPE BYTE,
= HOME$EXCHANGE ADDRESS,
= RESPONSE$EXCHANGE ADDRESS';
=
20 1 = DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
= MSG$HDR,
= REMAINDER (1) BYTE)';
=
21 1 declare timex exchange$descriptor external;
22 1 declare rqsrx exchange$descriptor external;
23 1 CQ$Q$INIT: Procedure (queue$ptr, queue$size) reentrant pub
- lic;
/*
/* This procedure initializes the queue pointers to
indicate an empty queue.
*/
24 2 Declare queue$ptr address;
25 2 Declare queue$size byte;
26 2 Declare queue based queue$ptr queue$format;
/* set up the size of the queue into the structure */
27 2 queue.FND$PTR = queue$size - 1;
/* reset the queue offset pointers */
28 2 queue.GIVE$PTR, queue.TAKE$PTR = 0;
/* return to calling program */
29 2 return;
end cq$~init;
30 2 end cq$init$module;
31 1

```

```

1      $title('NEXT GIVE QUEUE PROCEDURE, VERSION 2.1')
      NEXT$GIVE$MODULE: Do;

2      1      declare eom literally 'CDH';
      $include(:fl:commsg.elt)

3      1      =      declare msg$format literally 'structure (
      =      trgt      byte,
      =      cmd      byte,
      =      seq$la      byte,
      =      seq$na      byte,
      =      text(256) byte )';
      =

4      1      =      declare queue$format literally 'structure (
      =      end$ptr      byte,
      =      give$ptr      byte,
      =      take$ptr      byte,
      =      data$byte(254) byte )';
      =

5      1      =      declare par$format literally 'structure (
      =      la      byte,
      =      na      byte,
      =      cu      byte,
      =      stop      byte,
      =      que$pt      byte )';
      $include(:ff:synch.ext)

6      1      =      RQSEND:
      =      PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTFRNL;
7      2      =      DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
      =

8      2      =      END RQSEND;
      =

9      1      =      RQWAIT:
      =      PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
10     2      =      DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
      =

11     2      =      END RQWAIT;
      =

12     1      =      RQACPT:
      =      PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
13     2      =      DECLARE EXCHANGE$POINTER ADDRESS;
      =

14     2      =      END RQACPT;
      =

15     1      =      RQISND:
      =      PROCEDURE (IED$PTR) EXTERNAL;
16     2      =      DECLARE IED$PTR ADDRESS;
      =

17     2      =      END RQISND;
      $include(:ff:exch.elt)

18     1      =      DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      =      MSG$HEAD ADDRESS,
      =      MSG$TAIL ADDRESS,
      =      TASK$HEAD ADDRESS,
      =      TASK$TAIL ADDRESS,

```



```

= NEXT$EXCH ADDRESS);
$include(:fc:msg.elt)
19 1 = DECLARE MSG$HDR LITERALLY '
= LINK ADDRESS,
= LENGTH ADDRESS,
= TYPE BYTE,
= HOMF$EXCHANGE ADDRESS,
= RESPONSE$EXCHANGE ADDRESS';
=
20 1 = DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
= MSG$HDR,
= REMAINDER(1) BYTE)';
=
21 1 declare timex exchange$descriptor external;
22 1 declare rqfsrx exchange$descriptor external;
23 1 CQ$NEXT$GIVE: Procedure (QUEUE$PTR,GIVE$BYTE) byte reentran
- t public;
/*
This procedure places a byte into the queue if room
exists in the queue for that byte of data. If no room
exists, a queue full indicator will be returned by
the procedure.
*/
24 2 Declare QUEUE$FULL literally 'CFFH';
25 2 Declare QUEUE$OK literally 'C0CH';
26 2 Declare QUEUE$PTR address;
27 2 Declare (GIVE$BYTE,RSLT) byte;
28 2 declare queue based queue$ptr queue$format;
/* Test for queue full condition and if it is full,
then insert end of message */
29 2 RSLT = queue$ok;
30 2 If (queue.GIVE$PTR+1 > queue.END$PTR)
then do;
32 3 rslt = queue$full;
33 3 queue.data$byte(queue.give$ptr) = eom;
34 3 end;
35 2 else do;
/* Store the byte into the next queue location
- */
36 3 queue.DATA$BYTE(queue.GIVE$PTR) = GIVE$BYTE;
/* Increment the give pointer */
37 3 If ((queue.GIVE$PTR:=queue.GIVE$PTR+1)

```



```

Stitle('GET CHARACTER FROM QUEUE PROCEDURE')
1 NEXTSTAKESMODULE: Do;

2 1 declare eom literally 'EDH';
2 1 $include(:f1:commsg.elt)
3 1 = declare msg$format literally 'structure (
=      trgt      byte,
=      cmd      byte,
=      sec$la    byte,
=      sec$na    byte,
=      text(250) byte )';
=
4 1 = declare queue$format literally 'structure (
=      end$ptr    byte,
=      give$ptr   byte,
=      take$ptr   byte,
=      data$byte(254) byte )';
=
5 1 = declare par$format literally 'structure (
=      la        byte,
=      na        byte,
=      run       byte,
=      stop      byte,
=      que$pt    byte )';
$include(:f0:synch.ext)
6 1 = RQSEND:
=      PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
7 2 =      DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
=
8 2 =      END RQSEND;
=
9 1 = RQWAIT:
=      PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
10 2 =      DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
=
11 2 =      END RQWAIT;
=
12 1 = RQACPT:
=      PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
13 2 =      DECLARE EXCHANGE$POINTER ADDRESS;
=
14 2 =      END RQACPT;
=
15 1 = RQISND:
=      PROCEDURE (IED$PTR) EXTERNAL;
16 2 =      DECLARE IED$PTR ADDRESS;
=
17 2 =      END RQISND;
$include(:f0:exch.elt)
18 1 = DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
=      MSG$HEAD ADDRESS,
=      MSG$TAIL ADDRESS,
=      TASK$HEAD ADDRESS,
=      TASK$TAIL ADDRESS,

```

```

= NEXTSEXCH ADDRESS);
$include(:fc:msg.elt)
19 1 = DECLARE MSG$HDR LITERALLY '
= LINK ADDRESS,
= LENGTH ADDRESS,
= TYPE BYTE,
= HOME$EXCHANGE ADDRESS,
= RESPONSE$EXCHANGE ADDRESS';
=
20 1 = DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
= MSG$HDR,
= REMAINDER(1) BYTE)';
=
21 1 declare tinex exchange$descriptor external;
22 1 declare rqr$rx exchange$descriptor external;
23 1 CQ$NEXT$TAKE: Procedure (queue$ptr) byte reentrant public;
/*
This typed procedure gets the next byte from the
indicated queue and returns it to the calling
program. The queue take pointer is incremented.
*/
24 2 Declare queue$ptr address;
25 2 Declare take$byte byte;
26 2 declare queue based queue$ptr queue$format;
/* store next data byte in queue */
27 2 take$byte = queue.DATA$BYTE(queue.TAKE$PTR);
/* increment the take pointer to next location */
28 2 If ((queue.TAKE$PTR=queue.TAKE$PTR+1)
> queue.END$PTR)
then queue.TAKE$PTR = 0;
/* return the data byte to caller */
30 2 return take$byte;
31 2 end cq$next$take;
32 1 end next$take$module;

```

```

$title('ASCII to HEX CONVERSION PROCEDURE')
1  ASC$'MODULE: Do;
2  1  declare eom$format literally 'CDH';
3  1  $include(:f1:commsg.elt)
4  1  = declare msg$format literally 'structure (
5  =   trgt      byte,
6  =   cmd       byte,
7  =   seq$la    byte,
8  =   seq$na    byte,
9  =   text(250) byte )';
10 1  = declare queue$format literally 'structure (
11 =   end$ptr    byte,
12 =   give$ptr   byte,
13 =   take$ptr   byte,
14 =   data$byte(254) byte )';
15 1  = declare par$format literally 'structure (
16 =   la        byte,
17 =   na        byte,
18 =   run       byte,
19 =   stop      byte,
20 =   que$pt    byte )';
21 1  $include(:f2:synch.ext)
22 1  = RQSEND:
23 2  =   PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
24 2  =   DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
25 2  =
26 2  =   END RQSEND;
27 1  = RQWAIT:
28 2  =   PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
29 2  =   DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
30 2  =
31 2  =   END RQWAIT;
32 1  = RQACPT:
33 2  =   PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
34 2  =   DECLARE EXCHANGE$POINTER ADDRESS;
35 2  =
36 2  =   END RQACPT;
37 1  = RQISND:
38 2  =   PROCEDURE (IED$PTR) EXTERNAL;
39 2  =   DECLARE IED$PTR ADDRESS;
40 2  =
41 2  =   END RQISND;
42 1  $include(:f3:exch.elt)
43 1  = DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
44 =   MSG$HEAD ADDRESS,
45 =   MSG$TAIL ADDRESS,
46 =   TASK$HEAD ADDRESS,
47 =   TASK$TAIL ADDRESS,

```

```

= NEXT$EXCH ADDRESS)';
$include(:f0:msg.elt)
19 1 = DECLARE MSG$HDR LITERALLY '
= LINK ADDRESS,
= LENGTH ADDRESS,
= TYPE BYTE,
= HOME$EXCHANGE ADDRESS,
= RESPONSE$EXCHANGE ADDRESS';
=
20 1 = DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
= MSG$HDR,
= .A.I.A.I.ND(1) BYTE)';
21 1 declare timex exchange$descriptor external;
22 1 declare rqlsrx exchange$descriptor external;
23 1 cq$next$take:
procedure (q$ptr) byte external;
24 2 declare q$ptr address;
25 2 end cq$next$take;
26 1 /* CQ$ASC$HEX: Procedure(q$ptr,msg$ptr) reentrant public;
/*
This procedure transforms ASCII data in the input
queue into a hex string in the system message storage
area.
*/
27 2 Declare (nchar,char,n) byte;
28 2 Declare (q$ptr,msg$ptr) address;
29 2
Declare msg based msg$ptr msg$format;
/* throw away the start of message */
30 2 char = cq$next$take(q$ptr);
/* convert message target address */
31 2 char = cq$next$take(q$ptr);
32 2 nchar = cq$next$take(q$ptr);
33 2 if char > 40h
then char = char - 37h;
35 2 else char = char - 30h;
36 2 if nchar > 40h
then nchar = nchar - 37h;
38 2 else nchar = nchar - 30h;
39 2 msg.trgt = shl(char,4) + nchar;
/* convert command byte */
40 2 char = cq$next$take(q$ptr);
41 2 if char > 40h
then char = char - 37h;
43 2 else char = char - 30h;

```



```

44 2      msg.cmd = char;
45 2      /* convert LA sequence */
46 2      char = cq$next$take(q$ptr);
47 2      if char > 40h
48 2      then char = char - 37h;
49 2      else char = char - 30h;
50 2      msg.seq$la = char;
51 2      /* convert NA sequence */
52 2      char = cq$next$take(q$ptr);
53 2      if char > 40h
54 2      then char = char - 37h;
55 2      else char = char - 30h;
56 2      msg.seq$na = char;
57 2      /* do operations until end of message */
58 2      n = 0;
59 2      Do while (char:=cq$next$take(q$ptr)) <> FOM;
60 2      /* get next ASCII character */
61 2      nchar = cq$next$take(q$ptr);
62 2      /* convert to hex format */
63 2      if char > 40h
64 2      then char = char - 37h;
65 2      else char = char - 30h;
66 2      if nchar > 40h
67 2      then nchar = nchar - 37h;
68 2      else nchar = nchar - 30h;
69 2      msg.text(n) = shl(char,4) + nchar;
70 2      n = n + 1;
71 2      end;
72 2      end cq$asc$hex;
73 1      end asc$hex$module;

```

```

$title('HEX TO ASCII CONVERSION PROCEDURE')
HEX$ASC$MODULE: Do;

1      1      'declare eom literally 'CDH';
2      1      $include(:ff:commsg.elt)
3      1      =      declare msg$format literally 'structure (
4      1      =      =      trgt      byte,
5      1      =      =      cmdnd    byte,
6      1      =      =      seq$la   byte,
7      1      =      =      seq$na   byte,
8      1      =      =      text(25C) byte)';
9      1      =      declare queue$format literally 'structure (
10     1      =      =      end$ptr    byte,
11     1      =      =      give$ptr   byte,
12     1      =      =      take$ptr   byte,
13     1      =      =      data$byte(254) byte)';
14     1      =      declare pars$format literally 'structure (
15     1      =      =      la         byte,
16     1      =      =      na         byte,
17     1      =      =      run        byte,
18     1      =      =      stop       byte,
19     1      =      =      que$pt     byte)';
20     1      =      $include(:ff:synch.ext)
21     1      =      ROSEND:
22     2      =      PROCEDURE (EXCHANGES$POINTER,MESSAGES$POINTER) EXTERNAL;
23     2      =      DECLARE (EXCHANGES$POINTER,MESSAGES$POINTER) ADDRESS;
24     2      =
25     2      =      END ROSEND;
26     2      =
27     2      =      RQWAIT:
28     2      =      PROCEDURE (EXCHANGES$POINTER,DELAY) ADDRESS EXTERNAL;
29     2      =      DECLARE (EXCHANGES$POINTER,DELAY) ADDRESS;
30     2      =
31     2      =      END RQWAIT;
32     2      =
33     2      =      RQACPT:
34     2      =      PROCEDURE (EXCHANGES$POINTER) ADDRESS EXTERNAL;
35     2      =      DECLARE EXCHANGES$POINTER ADDRESS;
36     2      =
37     2      =      END RQACPT;
38     2      =
39     2      =      RQISND:
40     2      =      PROCEDURE (IED$PTR) EXTERNAL;
41     2      =      DECLARE IED$PTR ADDRESS;
42     2      =
43     2      =      END RQISND;
44     2      =      $include(:ff:exch.elt)
45     1      =      DECLARE EXCHANGES$DESCRIPTOR LITERALLY 'STRUCTURE (
46     1      =      =      MSG$HEAD ADDRESS,
47     1      =      =      MSG$TAIL ADDRESS,
48     1      =      =      TASK$HEAD ADDRESS,
49     1      =      =      TASK$TAIL ADDRESS,

```

```

= NEXTSEXCH ADDRESS)';
$include(:ff:msg.elt)
19 1 = DECLARE MSG$HDR, LITERALLY '
= LINK ADDRESS,
= LENGTH ADDRESS,
= TYPE BYTE,
= HOME$EXCHANGE ADDRESS,
= RESPONSE$EXCHANGE ADDRESS';
20 1 = DECLARE MSG$DESCRIPTOR, LITERALLY 'STRUCTURE(
= MSG HDR,
= RE: :f(1), BYTE)';
21 1 declare timex exchange$descriptor external;
22 1 declare rqlsrx exchange$descriptor external;
23 1 cq$next$give:
= procedure(queue$ptr, data$byte) byte external;
24 2 declare queue$ptr address;
25 2 declare data$byte byte;
26 2 end cq$next$give;
27 1 CQ$HEX$ASC: Procedure (q$ptr, hex$byte) reentrant public;

/*
This procedure is used to convert a hex byte into two
ASCII characters and store them into the next two loc-
ations of the Q$OUT data area.
*/
28 2 Declare (high$byte, hex$byte, low$byte, status) byte;
29 2 Declare q$ptr address;
30 2 Declare q$OUT based q$ptr queue$format;

/* separate low and high order bits */
31 2 If (high$byte:=(shr(hex$byte,4) and (FH)) > 0
then high$byte = high$byte + 37H;
33 2 else high$byte = high$byte + 30H;
34 2 If (low$byte:=(hex$byte and (FH)) > 0
then low$byte = low$byte + 37H;
36 2 else low$byte = low$byte + 30H;

/* store ASCII conversions into the queue */
37 2 status = cq$next$give(.Q$OUT, high$byte);
38 2 status = cq$next$give(.Q$OUT, low$byte);
39 2 return;
40 2 end cq$hex$asc;
41 1 end hex$asc$module;

```

```

$stitle('CHECKSUM CALCULATION PROCEDURE')
1 CHECKSUM$MODULE: Do;
2 1 declare eom literally 'EOM';
  $include(:f1:commsg.elt)
3 1 = declare msg$format literally 'structure (
  =   trgt      byte,
  =   cmd      byte,
  =   seq$la    byte,
  =   seq$na    byte,
  =   text(250) byte )';
4 1 = declare queue$format literally 'structure (
  =   end$ptr    byte,
  =   give$ptr   byte,
  =   take$ptr   byte,
  =   data$byte(254) byte )';
5 1 = declare par$format literally 'structure (
  =   la         byte,
  =   na         byte,
  =   run        byte,
  =   stop       byte,
  =   que$pt     byte )';
  $include(:f0:synch.ext)
6 1 = RQSEND:
  =   PROCEDURE (EXCHANGE$POINTER, MESSAGE$POINTER) EXTERNAL;
7 2 =   DECLARE (EXCHANGE$POINTER, MESSAGE$POINTER) ADDRESS;
8 2 =   END RQSEND;
9 1 = RQWAIT:
  =   PROCEDURE (EXCHANGE$POINTER, DELAY) ADDRESS EXTERNAL;
10 2 =   DECLARE (EXCHANGE$POINTER, DELAY) ADDRESS;
11 2 =   END RQWAIT;
12 1 = RQACPT:
  =   PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
13 2 =   DECLARE EXCHANGE$POINTER ADDRESS;
14 2 =   END RQACPT;
15 1 = RQISND:
  =   PROCEDURE (IED$PTR) EXTERNAL;
16 2 =   DECLARE IED$PTR ADDRESS;
17 2 =   END RQISND;
  $include(:f0:exch.elt)
18 1 = DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
  =   MSG$HEAD ADDRESS,
  =   MSG$TAIL ADDRESS,
  =   TASK$HEAD ADDRESS,
  =   TASK$TAIL ADDRESS,

```

```

= NEXT$EXCH ADDRESS);
$include(:ff:msg.elt)
19 1 = DECLARE MSG$HDR LITERALLY '
= LINK ADDRESS,
= LENGTH ADDRESS,
= TYPE BYTE,
= HOME$EXCHANGE ADDRESS,
= RESPONSE$EXCHANGE ADDRESS';
=
20 1 = DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE (
= MSG$HDR,
= REMAINDER(1) BYTE)';
21 1 declare timex exchange$descriptor external;
22 1 declare rqr$rx exchange$descriptor external;
23 1 CQ$CHECKSUM: Procedure(q$ptr) byte reentrant public;

/*
This procedure is used to determine the checksum of
a message which has been received and stored into
the Q$in buffer. A checksum will be calculated on all
characters beginning at the start of text and
continuing until the checksum bytes are encountered.
This value will be tested with the value transmitted
and if they agree, a value of zero will be returned
to the calling program. If not in agreement, a
non-zero value will be returned.
*/
24 2 Declare (ptr,end$ptr,q$ptr) address;
25 2 Declare (checksm,string$sum,char,n) byte;
26 2 Declare last$chars(3) byte;
27 2 declare q$in based q$ptr queue$format;

/* get queue pointer values for reference */
28 2 PTR = Q$IN.take$ptr;
29 2 END$PTR = Q$IN.end$ptr;

/* initialize checksum value */
30 2 checksm = 0;

/* compute checksum of string */
31 2 char = Q$IN.data$byte(ptr);
32 2 do while char <> EOM;
33 3 checksm = checksm + char;
34 3 if ptr = end$ptr
35 3 then ptr = 0;
36 3 else ptr = ptr + 1;
37 3 char = Q$IN.data$byte(ptr);
38 3 end;

```

```

/* last three characters in the message are
not included in the checksum, so they must
be subtracted....*/

/* first remember the last queue location */
39 2      if ptr = end$ptr
41 2      then char = 0;
42 2      else char = ptr+1;
43 3      do n = 0 to 2;
44 3          checksum = checksum - (last$chars(n) :=
45 3              Q$IN.data$byte(ptr));
46 3          if ptr = 0
47 3          then ptr = end$ptr;
48 2          else ptr = ptr - 1;
49 2          end;
50 2          checksum = checksum + last$chars(0);

/* convert transmitted checksum into a hex value */
51 2      do n = 1 to 2;
52 3          if last$chars(n) > 40H
53 3          then last$chars(n) = last$chars(n) - 37H;
54 3          else last$chars(n) = last$chars(n) - 20H;
55 2          end;
56 2          string$sum = shl(last$chars(2),4) + last$chars(1);

/* test for validity of transmission */
57 2      if string$sum = checksum
58 2      then return 0;

/* if bad checksum, clear data queue of message */
59 2      else Q$IN.take$ptr = char;
60 2      return -1;

end ca$checksum;
end checksum$module;

```



```

1  $title('GENERATE READY MESSAGE PROCEDURE')
2  GENSRDY$MODULE: Do;
3  1  declare eom literally 'GDH';
4  1  $include(:f1:commsg.elt)
5  1  =  declare msg$format literally 'structure (
6  =  trgt      byte,
7  =  cmnd      byte,
8  =  seq$la    byte,
9  =  seq$na    byte,
10 =  text(250) byte)';
11 1  =  declare queue$format literally 'structure (
12 =  end$ptr    byte,
13 =  give$ptr   byte,
14 =  take$ptr   byte,
15 =  data$byte(254) byte)';
16 1  =  declare par$format literally 'structure (
17 =  la         byte,
18 =  na         byte,
19 =  run        byte,
20 =  stop       byte,
21 =  que$pt     byte)';
22 1  $include(:f0:synch.ext)
23 1  =  RQSEND:
24 =  PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
25 2  =  DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
26 2  =
27 2  =  END RQSEND;
28 1  =  RQWAIT:
29 =  PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
30 2  =  DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
31 2  =
32 2  =  END RQWAIT;
33 1  =  RQACPT:
34 =  PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
35 2  =  DECLARE EXCHANGE$POINTER ADDRESS;
36 2  =
37 2  =  END RQACPT;
38 1  =  RQISND:
39 =  PROCEDURE (IED$PTR) EXTERNAL;
40 2  =  DECLARE IED$PTR ADDRESS;
41 2  =
42 2  =  END RQISND;
43 1  =  $include(:f0:exch.elt)
44 1  =  DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
45 =  MSG$HEAD ADDRESS,
46 =  MSG$TAIL ADDRESS,
47 =  TASK$HEAD ADDRESS,
48 =  TASK$TAIL ADDRESS,

```

```

= NEXT$EXCH ADDRESS);
$include(:fc:msg.elt)
19 1 = DECLARE MSG$HDR LITERALLY
= LINK ADDRESS,
= LENGTH ADDRESS,
= TYPE BYTE,
= HOME$EXCHANGE ADDRESS,
= RESPONSE$EXCHANGE ADDRESS';
20 1 = DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
= MSG$HDR,
= REMAINDER(1) BYTE)';

21 1 declare timex exchange$descriptor external;
22 1 declare rqfsrx exchange$descriptor external;

23 1 cq$msg$out:
procedure (msg$size,q$ptr,par$ptr,msg$ptr) external;
24 2 declare msg$size byte;
25 2 declare (q$ptr,par$ptr,msg$ptr) address;
26 2 end cq$msg$out;

27 1 CQ$GEN$RDY: Procedure (trget,msg$ptr,q$ptr,par$ptr) reentr
- ant public;

/*
This procedure generates a "ready" message onto the
communication network. The target address is passed
as the parameter in the call.

On entry to the procedure:
trget is a byte which contains the address
of the target node.
msg$ptr is an address which points to
the RAM work area.
q$ptr is an address which points to
the output queue.
par$ptr is an address which points to
the communication flags.
*/

28 2 Declare trget byte;
29 2 Declare rdy$msg structure (
TARGET byte,
COMAND byte,
SEQ$LA byte,
SEQ$NA byte )
data ( 0,3,0,0 );

30 2 declare (msg$ptr,q$ptr,par$ptr) address;
31 2 declare msg based msg$ptr msg$format;

/* move format into message block */

```

```

32  2      call move (d, .rdy$msq, msq$ptr);
          /* insert current parameters */
33  2      msg.trgt = trgt;
          /* send message */
34  2      call cq$msq$ptr, par$ptr, par$ptr, msq$ptr;
35  2      return;
36  2      end cq$gen$rdy;
37  1      end gen$rdy;

declare tlmx exchange$descriptor external;
declare rdsx exchange$descriptor external;

cqlmsgout:
procedure (msgsize, dptr, par$ptr, msq$ptr) external;
declare msgsize byte;
declare (dptr, par$ptr, msq$ptr) address;
end cqlmsgout;

CQGENRDY: Procedure (trgt, msg$ptr, dptr, par$ptr) local;
    not public;

*
This procedure generates a "ready" message onto the
communication network. The target address is passed
as the parameter in the call.

On entry to the procedure:
    trgt is a byte which contains the address
    of the target node.
    msg$ptr is an address which points to
    the RAM work area.
    dptr is an address which points to
    the output queue.
    par$ptr is an address which points to
    the communication flags.

38  2      declare trgt byte;
39  2      declare rdy$msq structure (
          TARGET byte,
          COMMAND byte,
          SEQ$LA byte,
          SEQ$NA byte )
          data ( 0, 0, 0, 0 );
40  2      declare (msg$ptr, dptr, par$ptr) address;
41  2      declare msg base$rdy msg$format;
          /* move format into message block */

```

```

1          $title('DATA MESSAGE GENERATOR PROCEDURE')
GEN$M$C$MODULE: Do;

2      1      declare eom literally 'CDH';
$include(:f1:commmsg.elt)

3      1      =      declare msg$format literally 'structure (
=          trgt      byte,
=          cmd      byte,
=          seq$la      byte,
=          seq$na      byte,
=          text(250) byte )';

4      1      =      declare queue$format literally 'structure (
=          end$ptr      byte,
=          give$ptr      byte,
=          take$ptr      byte,
=          data$byte(254) byte )';

5      1      =      declare par$format literally 'structure (
=          la      byte,
=          na      byte,
=          run      byte,
=          stop      byte,
=          que$pt      byte )';
$include(:f0:synch.ext)

6      1      =      RQSEND:
=          PROCEDURE (EXCHANGE$POINTER, MESSAGE$POINTER) EXTERNAL;
7      2      =      DECLARE (EXCHANGE$POINTER, MESSAGE$POINTER) ADDRESS;
=
8      2      =      END RQSEND;

9      1      =      RQWAIT:
=          PROCEDURE (EXCHANGE$POINTER, DELAY) ADDRESS EXTERNAL;
10     2      =      DECLARE (EXCHANGE$POINTER, DELAY) ADDRESS;
=
11     2      =      END RQWAIT;

12     1      =      RQACPT:
=          PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
13     2      =      DECLARE EXCHANGE$POINTER ADDRESS;
=
14     2      =      END RQACPT;

15     1      =      RQISND:
=          PROCEDURE (IED$PTR) EXTERNAL;
16     2      =      DECLARE IED$PTR ADDRESS;
=
17     2      =      END RQISND;

18     1      =      $include(:f0:exch.elt)
=      DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
=          MSG$HEAD ADDRESS,
=          MSG$TAIL ADDRESS,
=          TASK$HEAD ADDRESS,
=          TASK$TAIL ADDRESS,

```

```

= NEXT$EXCH ADDRESS);
$include(:ff:msg.elt)
19 1 = DECLARE MSG$HDR LITERALLY '
= LINK ADDRESS,
= LENGTH ADDRESS,
= TYPE BYTE,
= HOME$EXCHANGE ADDRESS,
= RESPONSE$EXCHANGE ADDRESS';
=
20 1 = DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE (
= MSG$HDR,
= REMAINDER(1) BYTE)';
21 1 declare msg$exchange$descriptor external;
22 1 declare rq$rx$exchange$descriptor external;
23 1 cq$msg$out:
= procedure (msg$size, q$ptr, par$ptr, msg$ptr) external;
24 2 declare msg$size byte;
25 2 declare (q$ptr, par$ptr, msg$ptr) address;
26 2 end cq$msg$out;
27 1 CQ$GEN$MSG: Procedure (msg$pointer, trget, msg$ptr, q$ptr,
- par$ptr, save$flg) reentrant public;
/*
This procedure generates a data message and places
it onto the system communications network.

On entry to the procedure:
msg$pointer is an address which points to
the data to be transmitted.
trget is an byte which contains the
address of the target node.
msg$ptr is an address which points to
the RAM work area.
q$ptr is an address which points to
the output queue.
par$ptr is an address which points to
the communications flags.
*/
28 2 - Declare (msg$pointer, msg$ptr, q$ptr, par$ptr, ram$size) a
address;
29 2 - Declare (trget, save$flg) byte;
30 2 Declare msg based msg$ptr msg$format;
31 2 Declare data$msg structure (
target byte,
command byte,
seq$LA byte,
seq$NA byte,
text byte )
data (0, 2, 0, 0, 0);

```

```

32      2 declare data$block based msg$pointer structure (
          msg$hdr );
          /* move message format into message buffer */

33      2 ***** call move (5, .data$msg, msg$ptr); *****

          /* insert target address */
34      2 msg.trgt = trgt;
          /* append data to communication message */

35      2 call move (data$block.length, msg$pointer, .msg.text(0
          -msg));
          /* transmit message onto network */

          *****
36      2 ram$size = data$block.length;
37      2 call cq$msg$out (ram$size, q$ptr, par$ptr, msg$ptr);

          /* return memory to free space manager */

38      2 if save$flg
          then do;
39      3 if ram$size mod 4 > 0
          then data$block.length = data$block.length + (4 - (r
          - ram$size mod 4));
40      3 call RQSEND (.rqf$rx, msg$pointer);
41      3 end;

42      2 return;
43      2 end cq$gen$msg;
44      1 end gen$msg$module;

```



```

) title('ISBX 351 Communications Driver, Version 1.3')
$nointvector
1  CQS351: Do;

/*****

This module contains the physical interface for
support of the Intel ISBX 351 communications expansion
board. The board is inserted into socket J6 and has a
base address of FCH with the interrupt from the
receiver strapped to the interrupt level 6. The
transmitter interrupt is wired to interrupt level 5.

Multi-drop communication options are supported by
the physical software package.

*****/

$include (:f0:exch.elt)
2  1  =  DECLARE EXCHANGES$DESCRIPTOR LITERALLY 'STRUCTURE (
      =  MSG$HEAD ADDRESS,
      =  MSG$TAIL ADDRESS,
      =  TASK$HEAD ADDRESS,
      =  TASK$TAIL ADDRESS,
      =  NEXT$EXCH ADDRESS);
3  1  =  $include (:f0:ied.elt)
      =  DECLARE INT$EXCHANGES$DESCRIPTOR LITERALLY 'STRUCTURE (
      =  MESSAGE$HEAD ADDRESS,
      =  MESSAGE$TAIL ADDRESS,
      =  TASK$HEAD ADDRESS,
      =  TASK$TAIL ADDRESS,
      =  EXCHANGES$LINK ADDRESS,
      =  LINK ADDRESS,
      =  LENGTH ADDRESS,
      =  TYPE BYTE);
4  1  =  Declare RQL6EX int$exchange$descriptor external;
5  1  =  Declare RQL7EX int$exchange$descriptor external;

$include (:f1:commmsg.elt)
6  1  =  declare msg$format literally 'structure (
      =  trgt      byte,
      =  cmnd      byte,
      =  seq$la     byte,
      =  seq$na     byte,
      =  text(250) byte );
7  1  =  declare queue$format literally 'structure (
      =  end$ptr     byte,
      =  give$ptr    byte,
      =  take$ptr    byte,
      =  data$byte(254) byte );
8  1  =  declare par$format literally 'structure (

```

```

101 1 = Declare (msg$pointer,byte,byte);
102 1 = end cmsg$ptr;
103 1 = run byte;
104 1 = stop byte;
105 1 = cmsg$ptr;
106 1 = PROCEDURE (EXCHANGE);
107 1 = DECLARE CQINSL queue$format external;
108 1 = Declare CQCUTSL queue$format external;
109 1 = Declare CQPARS par$format external;
110 1 = $include (:fl:comcom.ext);
111 1 = CQ$NEXT$GIVE:
112 1 = Procedure (q$ptr,give$byte) byte external;
113 2 = Declare q$ptr address;
114 2 = Declare give$byte byte;
115 2 = end cmsg$next$give;
116 1 = CQ$TAKE$TEST:
117 2 = Procedure (q$ptr) byte external;
118 2 = Declare q$ptr address;
119 2 = end cmsg$take$test;
120 1 = CQ$NEXT$TAKE:
121 2 = Procedure (q$ptr) byte external;
122 2 = Declare q$ptr address;
123 2 = end cmsg$next$take;
124 1 = CQ$Q$INIT:
125 2 = Procedure (q$ptr,q$size) external;
126 2 = Declare q$ptr address;
127 2 = Declare q$size byte;
128 2 = end cmsg$q$init;
129 1 = CQ$ASC$HEX:
130 2 = Procedure (q$ptr,msg$ptr) external;
131 2 = Declare (q$ptr,msg$ptr) address;
132 2 = end cmsg$asc$hex;
133 1 = CQ$CHECK$SUM:
134 2 = Procedure (q$ptr) byte external;
135 2 = Declare q$ptr address;
136 2 = end cmsg$check$sum;
137 1 = CQ$HEX$ASC:
138 2 = Procedure (q$ptr,hex$byte) external;
139 2 = Declare q$ptr address;
140 2 = Declare hex$byte byte;
141 2 = end cmsg$hex$asc;
142 1 = CQ$MSG$OUT:
143 2 = Procedure (msg$size,q$ptr,par$ptr,msg$ptr) external;
144 2 = Declare msg$size byte;
145 2 = Declare (q$ptr,par$ptr,msg$ptr) address;
146 2 = end cmsg$msg$out;
147 1 = CQ$GEN$RDY:
148 2 = Procedure (trget,msg$ptr,q$ptr,par$ptr) external;
149 2 = Declare trget byte;
150 2 = Declare (msg$ptr,q$ptr,par$ptr) address;
151 2 = end cmsg$gen$rdy;
152 1 = CQ$GEN$MSG:
153 2 = Procedure (msg$pointer,trget,msg$ptr,q$ptr,par$ptr,save$flag) external;
154 2 = Declare (trget,save$flag) byte;

```

```

46 2 =      Declare (msg$pointer,msg$ptr,q$ptr,par$ptr) address;
47 2 =      end cq$gen$msg;
      $include (:fc:synch.ext)
48 1 =      RQSEND:
      =      PROCEDURE (EXCHANGES$POINTER,MESSAGE$POINTER) EXTERNAL;
49 2 =      DECLARE (EXCHANGES$POINTER,MESSAGE$POINTER) ADDRESS;
      =
50 2 =      END RQSEND;
      =
51 1 =      RQWAIT:
      =      PROCEDURE (EXCHANGES$POINTER,DELAY) ADDRESS EXTERNAL;
52 2 =      DECLARE (EXCHANGES$POINTER,DELAY) ADDRESS;
      =
53 2 =      END RQWAIT;
      =
54 1 =      RQACPT:
      =      PROCEDURE (EXCHANGES$POINTER) ADDRESS EXTERNAL;
55 2 =      DECLARE EXCHANGES$POINTER ADDRESS;
      =
56 2 =      END RQACPT;
      =
57 1 =      RQISND:
      =      PROCEDURE (IED$PTR) EXTERNAL;
58 2 =      DECLARE IED$PTR ADDRESS;
      =
59 2 =      END RQISND;
      $include (:fc:intrpt.ext)
60 1 =      RQENDI:
      =      PROCEDURE EXTERNAL;
      =
61 2 =      END RQENDI;
      =
62 1 =      RQELVL:
      =      PROCEDURE (LEVEL) EXTERNAL;
63 2 =      DECLARE LEVEL BYTE;
      =
64 2 =      END RQELVL;
      =
65 1 =      RQDLVL:
      =      PROCEDURE (LEVEL) EXTERNAL;
66 2 =      DECLARE LEVEL BYTE;
      =
67 2 =      END RQDLVL;
      =
68 1 =      RQSETV:
      =      PROCEDURE (PROC,LEVEL) EXTERNAL;
69 2 =      DECLARE PROC ADDRESS;
70 2 =      DECLARE LEVEL BYTE;
      =
71 2 =      END RQSETV;
      =
72 1 =      RQSETP:
      =      PROCEDURE (PROC,LEVEL) EXTERNAL;
73 2 =      DECLARE PROC ADDRESS;
74 2 =      DECLARE LEVEL BYTE;

```

```

75 2 =      END RQSETP;

76 1      Declare EOM literally 'CDH';
          /* defines end of message */
77 1      Declare RTS literally '00100000b';
          /* ready to send to USART */
78 1      Declare RXE literally '00000100b';
          /* ready to receive to USART */
79 1      Declare DTR literally '00000010b';
          /* data set ready to USART */
80 1      Declare TXEN literally '00000001b';
          /* transmit enable to USART */
81 1      Declare badint byte;
          /* flag for interrupt 5 */

          $eject
          /*****
          This procedure initializes the hardware required to
          operate the iSPX 351 expansion board.
          *****/

82 1      CQ$INIT$351:
          Procedure public;
83 2      Declare n byte;

          /* initialize the timer/counters */
84 2      output(0fbh) = 0b6h;          /* select counter 2 */
85 2      output(0fah) = 32;          /* lsb for 2400 baud */
86 2      output(0fah) = 0;          /* msb for 2400 baud */

          /* initialize the USART */
87 2      output(0flh) = 80h;          /* clear out garbage */
88 2      output(0flh) = 0;          /* ... more garbage */
89 2      output(0flh) = 40h;          /* reset the USART */
90 2      output(0flh) = 00eh;          /* 8 bit, no parity, 2 st
          - op */
91 2      output(0flh) = 26h;          /* enable receive mode */

          /* tell the nucleus the vector location */

92 2      CALL rqsetv(.cqmivt$351, 6);
93 2      call rsetv(.send$char$351, 5);
94 2      call rqlvl( 5);
95 2      call rqlvl(6);
96 2      return;
97 2      end cq$init$351;

          $eject
          /*****

```

This procedure stores a character from the USART into the receiver data input queue.

```

*****/
98 1  CQMIVT$351:
99 2      Procedure interrupt 6 public;
      Declare (GIVE$STATUS,CHAR) byte;

      /* get character from USART */
100 2      char = input($F0h) and 07Fh;
101 2      give$status = cq$next$give(.cqinsl, char);
102 2      if char = eom
      then call rqisnd(.rqisnex);
104 2      else call rqendi;

105 2      end cqmivt$351;
      $eject
      /*****
      This procedure sends out the next character to the
      selected USART device. It will disable the interrupt
      when all desired characters have been transmitted.
      *****/
106 1  SEND$CHAR$351:
107 2      Procedure interrupt 5 public;
      Declare char byte;

      /*testing if interrupt is really for 5 and not noise*/

108 2      if badint > 0 then do;

      /* enable drivers at beginning of message */
110 3      If not cqpars.run
      then do;
112 4          cqpars.run = 1;
113 4          cqpars.stop = 0;
114 4      end;

      /* disable drivers at end of message */
115 3      If cqpars.stop
      then do;
117 4          cqpars.run = 0;
118 4          Do while (input($F1h) and 4) = 0;
119 5              end;
120 4          badint = 0;
121 4          output($F1h) = 26h;
122 4      end;

      /* if message is in progress, send next character */
123 3      else do;
124 4          char = cq$next$take(.cqouts1);
125 4          do while (input($F1h) and 4) = 0;
126 5              end;
127 4          output($F0h) = char;

      /* test for end of message */
128 4          if (char and 0fh) = eom

```

```

130 4      then cqpars.stop = 1;
131 4      else cqpars.stop = 0;
132 2      end;
132 2      end;

      /* re-enable interrupts */
133 2      call rqendi;

134 2      return;

135 2      end send$char$351;
      $reject
      /*****

This procedure begins the transmission to a selected node.

*****/
136 1      CQ$START$MSG$351:
      Procedure public;

137 2      badint = 0ffh;
138 2      output(0F1h) = 25h;
139 2      return;
140 2      end cq$start$msg$351;

141 1      end cqs351;

```



```

130 4 then cpgars.stop = 1;
131 4 else cpgars.stop = 0;
132 4 end;
133 4 end;
134 4 /* re-enable interrupts */
135 4 call redbit;
136 4 return;
137 4 end sendchar32;
138 4
139 4
140 4
141 4
142 4
143 4
144 4
145 4
146 4
147 4
148 4
149 4
150 4
151 4
152 4
153 4
154 4
155 4
156 4
157 4
158 4
159 4
160 4
161 4
162 4
163 4
164 4
165 4
166 4
167 4
168 4
169 4
170 4
171 4
172 4
173 4
174 4
175 4
176 4
177 4
178 4
179 4
180 4
181 4
182 4
183 4
184 4
185 4
186 4
187 4
188 4
189 4
190 4
191 4
192 4
193 4
194 4
195 4
196 4
197 4
198 4
199 4
200 4
201 4
202 4
203 4
204 4
205 4
206 4
207 4
208 4
209 4
210 4
211 4
212 4
213 4
214 4
215 4
216 4
217 4
218 4
219 4
220 4
221 4
222 4
223 4
224 4
225 4
226 4
227 4
228 4
229 4
230 4
231 4
232 4
233 4
234 4
235 4
236 4
237 4
238 4
239 4
240 4
241 4
242 4
243 4
244 4
245 4
246 4
247 4
248 4
249 4
250 4
251 4
252 4
253 4
254 4
255 4
256 4
257 4
258 4
259 4
260 4
261 4
262 4
263 4
264 4
265 4
266 4
267 4
268 4
269 4
270 4
271 4
272 4
273 4
274 4
275 4
276 4
277 4
278 4
279 4
280 4
281 4
282 4
283 4
284 4
285 4
286 4
287 4
288 4
289 4
290 4
291 4
292 4
293 4
294 4
295 4
296 4
297 4
298 4
299 4
300 4
301 4
302 4
303 4
304 4
305 4
306 4
307 4
308 4
309 4
310 4
311 4
312 4
313 4
314 4
315 4
316 4
317 4
318 4
319 4
320 4
321 4
322 4
323 4
324 4
325 4
326 4
327 4
328 4
329 4
330 4
331 4
332 4
333 4
334 4
335 4
336 4
337 4
338 4
339 4
340 4
341 4
342 4
343 4
344 4
345 4
346 4
347 4
348 4
349 4
350 4
351 4
352 4
353 4
354 4
355 4
356 4
357 4
358 4
359 4
360 4
361 4
362 4
363 4
364 4
365 4
366 4
367 4
368 4
369 4
370 4
371 4
372 4
373 4
374 4
375 4
376 4
377 4
378 4
379 4
380 4
381 4
382 4
383 4
384 4
385 4
386 4
387 4
388 4
389 4
390 4
391 4
392 4
393 4
394 4
395 4
396 4
397 4
398 4
399 4
400 4
401 4
402 4
403 4
404 4
405 4
406 4
407 4
408 4
409 4
410 4
411 4
412 4
413 4
414 4
415 4
416 4
417 4
418 4
419 4
420 4
421 4
422 4
423 4
424 4
425 4
426 4
427 4
428 4
429 4
430 4
431 4
432 4
433 4
434 4
435 4
436 4
437 4
438 4
439 4
440 4
441 4
442 4
443 4
444 4
445 4
446 4
447 4
448 4
449 4
450 4
451 4
452 4
453 4
454 4
455 4
456 4
457 4
458 4
459 4
460 4
461 4
462 4
463 4
464 4
465 4
466 4
467 4
468 4
469 4
470 4
471 4
472 4
473 4
474 4
475 4
476 4
477 4
478 4
479 4
480 4
481 4
482 4
483 4
484 4
485 4
486 4
487 4
488 4
489 4
490 4
491 4
492 4
493 4
494 4
495 4
496 4
497 4
498 4
499 4
500 4
501 4
502 4
503 4
504 4
505 4
506 4
507 4
508 4
509 4
510 4
511 4
512 4
513 4
514 4
515 4
516 4
517 4
518 4
519 4
520 4
521 4
522 4
523 4
524 4
525 4
526 4
527 4
528 4
529 4
530 4
531 4
532 4
533 4
534 4
535 4
536 4
537 4
538 4
539 4
540 4
541 4
542 4
543 4
544 4
545 4
546 4
547 4
548 4
549 4
550 4
551 4
552 4
553 4
554 4
555 4
556 4
557 4
558 4
559 4
560 4
561 4
562 4
563 4
564 4
565 4
566 4
567 4
568 4
569 4
570 4
571 4
572 4
573 4
574 4
575 4
576 4
577 4
578 4
579 4
580 4
581 4
582 4
583 4
584 4
585 4
586 4
587 4
588 4
589 4
590 4
591 4
592 4
593 4
594 4
595 4
596 4
597 4
598 4
599 4
600 4
601 4
602 4
603 4
604 4
605 4
606 4
607 4
608 4
609 4
610 4
611 4
612 4
613 4
614 4
615 4
616 4
617 4
618 4
619 4
620 4
621 4
622 4
623 4
624 4
625 4
626 4
627 4
628 4
629 4
630 4
631 4
632 4
633 4
634 4
635 4
636 4
637 4
638 4
639 4
640 4
641 4
642 4
643 4
644 4
645 4
646 4
647 4
648 4
649 4
650 4
651 4
652 4
653 4
654 4
655 4
656 4
657 4
658 4
659 4
660 4
661 4
662 4
663 4
664 4
665 4
666 4
667 4
668 4
669 4
670 4
671 4
672 4
673 4
674 4
675 4
676 4
677 4
678 4
679 4
680 4
681 4
682 4
683 4
684 4
685 4
686 4
687 4
688 4
689 4
690 4
691 4
692 4
693 4
694 4
695 4
696 4
697 4
698 4
699 4
700 4
701 4
702 4
703 4
704 4
705 4
706 4
707 4
708 4
709 4
710 4
711 4
712 4
713 4
714 4
715 4
716 4
717 4
718 4
719 4
720 4
721 4
722 4
723 4
724 4
725 4
726 4
727 4
728 4
729 4
730 4
731 4
732 4
733 4
734 4
735 4
736 4
737 4
738 4
739 4
740 4
741 4
742 4
743 4
744 4
745 4
746 4
747 4
748 4
749 4
750 4
751 4
752 4
753 4
754 4
755 4
756 4
757 4
758 4
759 4
760 4
761 4
762 4
763 4
764 4
765 4
766 4
767 4
768 4
769 4
770 4
771 4
772 4
773 4
774 4
775 4
776 4
777 4
778 4
779 4
780 4
781 4
782 4
783 4
784 4
785 4
786 4
787 4
788 4
789 4
790 4
791 4
792 4
793 4
794 4
795 4
796 4
797 4
798 4
799 4
800 4
801 4
802 4
803 4
804 4
805 4
806 4
807 4
808 4
809 4
810 4
811 4
812 4
813 4
814 4
815 4
816 4
817 4
818 4
819 4
820 4
821 4
822 4
823 4
824 4
825 4
826 4
827 4
828 4
829 4
830 4
831 4
832 4
833 4
834 4
835 4
836 4
837 4
838 4
839 4
840 4
841 4
842 4
843 4
844 4
845 4
846 4
847 4
848 4
849 4
850 4
851 4
852 4
853 4
854 4
855 4
856 4
857 4
858 4
859 4
860 4
861 4
862 4
863 4
864 4
865 4
866 4
867 4
868 4
869 4
870 4
871 4
872 4
873 4
874 4
875 4
876 4
877 4
878 4
879 4
880 4
881 4
882 4
883 4
884 4
885 4
886 4
887 4
888 4
889 4
890 4
891 4
892 4
893 4
894 4
895 4
896 4
897 4
898 4
899 4
900 4
901 4
902 4
903 4
904 4
905 4
906 4
907 4
908 4
909 4
910 4
911 4
912 4
913 4
914 4
915 4
916 4
917 4
918 4
919 4
920 4
921 4
922 4
923 4
924 4
925 4
926 4
927 4
928 4
929 4
930 4
931 4
932 4
933 4
934 4
935 4
936 4
937 4
938 4
939 4
940 4
941 4
942 4
943 4
944 4
945 4
946 4
947 4
948 4
949 4
950 4
951 4
952 4
953 4
954 4
955 4
956 4
957 4
958 4
959 4
960 4
961 4
962 4
963 4
964 4
965 4
966 4
967 4
968 4
969 4
970 4
971 4
972 4
973 4
974 4
975 4
976 4
977 4
978 4
979 4
980 4
981 4
982 4
983 4
984 4
985 4
986 4
987 4
988 4
989 4
990 4
991 4
992 4
993 4
994 4
995 4
996 4
997 4
998 4
999 4
1000 4

```


Using the RMX 86™ Operating System on iAPX 86™ Component Designs

Contents

INTRODUCTION	2-245
OPERATING SYSTEMS FUNCTIONAL OVERVIEW	
Multiprogramming	2-246
Multitasking	2-246
Growth	2-246
Scheduling	2-246
Communication and Synchronization ...	2-247
Resource Management	2-247
Interrupt Management	2-247
Initialization	2-248
Debugging	2-248
Higher Level Functions	2-248
Extendibility	2-248
iRMX 86 OPERATING SYSTEM ARCHITECTURE	
Layers	2-248
Configurability	2-249
Support Functions	2-249
Object Orientation	2-249
Task Scheduling	2-250
System Hardware Requirements	2-251
APPLICATION EXAMPLE	2-251
System Design	2-252
Jobs and Tasks	2-253
Interfaces and Synchronization	2-254
Priority	2-255
APPLICATION IMPLEMENTATION ...	2-255
Display Functions	2-255
Cataloging	2-256
Application Code	2-256
Supervisor Task	2-256
Input Task	2-256
FFT Task	2-257
Output Task	2-257
Terminal Handler	2-257
Nucleus Calls	2-258
System Configuration	2-258
SUMMARY	2-260
APPENDIX A — Supervisor Task Listings	
	2-262
APPENDIX B — Input Task Listings	
	2-283
APPENDIX C — System Configuration ...	
	2-300

Users with a wide range of applications will find the iRMX 86 Operating System allows them to implement a corresponding range of capabilities, from a minimum iRMX 86 Nucleus to a high level human interface. A complete iRMX 86 Operating System includes extensive I/O capabilities, debugger, application loader, bootstrap loader, and integrated user functions. This flexibility allows one operating system to be used for many projects, minimizing software learning curves for new applications.

This note discusses a relatively small standalone spectrum analysis system based on a subset of the iRMX 86 Nucleus. The intent of the note is to demonstrate advantages of using operating systems in hardware component designs. An overview of operating system functions is given first as background information. Readers familiar with operating systems may wish to skip this section. The overview is followed by a summary of the iRMX 86 Operating System. The summary is brief, as only the iRMX 86 Nucleus is used in this application. A detailed discussion may be found in Application Note AP-86, "Using the iRMX 86 Operating System," and iRMX 86 System Manuals.

The spectrum analysis system is described after the summary. The system requirements, design and implementation are detailed. The system software is discussed next, followed by configuration and hardware implementation. A summary completes the application note text. Partial code listings of the system software are included in the appendices.

OPERATING SYSTEMS FUNCTIONAL OVERVIEW

Operating system software manage initialization, resources, scheduling, synchronization, and protection of tasks or functions within the system, as well as providing facilities for maintenance, debugging, and growth. In general, operating systems support many of the following:

Multiprogramming

Multiprogramming provides the capability for two or more programs to share the system hardware, after being developed and implemented independently. Within the environment of an operating system, the programs are called jobs. Jobs include system resources, such as memory, in addition to the actual program code. Multiprogramming allows jobs that are required only during development, such as debuggers, to run in the target system. When development is completed, these jobs are removed from the final system without affecting the integrity of the remaining jobs.

Multitasking

Multitasking allows functions within a job to be handled by separate tasks. This is particularly valuable when a job is responsible for multiple asynchronous events or activities. One task can be assigned to each event or activity. Tasks are the functional members of the system, executing within the bounds of a job environment. Program code for a multitasking system is modular, with well-defined interfaces and communication protocols. The modular boundaries serve several important purposes. The code for each module can be generated and tested independent of the other modules. In addition, the boundaries confine errors, speeding debugging and simplifying maintenance.

Growth

The modular independence that results from multiprogramming and multitasking gives users the ability to efficiently create new applications by adding functions to old software. Applications can be tailored to specific needs by integrating new modules with previously-written general support code. If care is taken in system design, functions can be added in the field. Documentation for the older software can be carried on to the new applications. This growth path will save completely re-writing expensive custom software for each new application.

Scheduling

Even though a system has multiple jobs and tasks, only one task is actually running on the central processor at any single point in time. Scheduling provides a means of predicting and controlling the selection of the running task from the tasks that are ready to run. Basic scheduling methods include preemptive priority, non-preemptive priority, time-slice, and round-robin. Batch systems often use non-preemptive priority scheduling, in which the highest priority job gets control of the central processor and runs to completion. Preemptive scheduling is typically used in real-time or event-driven systems, where dedicated, quick response is the main concern. A higher-priority waiting task that becomes ready to run will preempt the lower-priority running task. Priority may be either set at task creation (static) or modified during running of the task (dynamic).

Time-slice and round-robin scheduling are used in multiuser or multitask systems that share processing resources and have limits on maximum execution time. Time-slice scheduling gives each task or job a fixed slice of dedicated processor time. Round-robin gives each task or job a turn at using the processor. The time available during the turn depends on system load and task priority.

Communication and Synchronization

Jobs and tasks in a multiprogramming and multitasking environment require a structured means of communication. This communication may be necessary to synchronize processes or to pass data between processes. Two means of providing communication are mailboxes and semaphores. Mailboxes are an exchange place for system messages. The messages may include data or provide access to other system objects, including other mailboxes. Tasks can send objects to a mailbox or wait for objects from a mailbox. Generally, the task has the option of waiting for a specified period of time for the message. The wait time may range from zero for a task that requires immediate response to infinite time for a task that must have a message to continue processing. Multiple mailboxes are used to synchronize multiple tasks.

A communication flow using mailboxes is shown in Figure 2. In this example, the sending task sends a message to a mailbox and specifies a return mailbox. The sending task then waits at the return mailbox. The receiving task obtains the message from the sending mailbox and sends a message to the return mailbox. The first task obtains the second message from the return mailbox, synchronizing the two tasks and passing data.

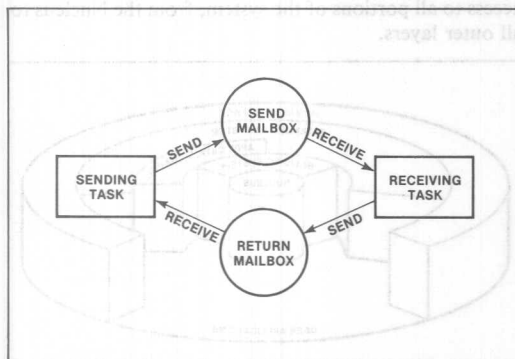


Figure 2. Intertask Communications with Mailboxes

If process synchronization is the only requirement, semaphores may be used. Semaphores function like mailboxes except that no data is passed through the semaphore. Instead, semaphores contain "units," with the meaning of the units defined by the sending and receiving tasks. A one unit semaphore may be used as a flag to synchronize the tasks. Multiple unit semaphores can be used for resource control. For example, if tasks require reusable data buffers, a semaphore may be defined as the allocator of the available data buffers. Each unit in the semaphore will represent one available buffer. When a task requires buffers to continue, the task will wait at the semaphore until enough units (representing

buffers) are available. The waiting task will receive the units, use the buffers, and return the units (still representing buffers) to the semaphore. Other tasks that require buffers will also have to wait at the semaphore until enough buffers are available.

Resource Management

The operating system is the central guardian of system resources, specifically read/write memory. The memory is made known to the Nucleus at initialization. The Nucleus then gives pieces or segments of the memory to tasks as they request it. This allows the tasks to have no initial knowledge of the actual location and size of system memory. Tasks can share memory if they desire, but the Nucleus allocates memory to each task individually, preventing the tasks from using each others memory. In addition, tasks return memory to the Nucleus when they are through with it, allowing memory to be reused.

Other system resources, such as I/O devices, will also be scheduled by the operating system. The operating system is responsible both for the efficient use of these resources and for providing the tasks with a large measure of independence from the actual I/O hardware requirements. The system may require many types of I/O devices, such as disk drives, tape drives, and printers. I/O is more efficiently accomplished if the operating system provides both asynchronous and synchronous I/O operations. Synchronous operations are those the task starts and waits for completion, doing no other work until the I/O is complete. Asynchronous operations are started by the task, but the actual I/O can take place while the task is doing other work. The overlapped operation of asynchronous I/O provides more user control of the I/O operation at the expense of a more complicated user interface.

Interrupt Management

Real-time software is tightly coupled to hardware functions by interrupts. Interrupts provide rapid notification that the hardware needs attention. The software must respond quickly without corrupting the system environment. System integrity is preserved by preempting the lower priority operating task, saving the task environment, processing the interrupt (including communicating the results to other tasks if necessary), restoring the environment of the operating task, and continuing. All of this must occur in an orderly and efficient manner. The interrupt management of the operating system is responsible for directly interacting with the system hardware that detects interrupts. The interrupt tasks can be ignorant of the detailed interrupt hardware, providing only the system actions to service the event that caused the interrupt.

Initialization

Operating systems create and manage jobs and tasks at initialization as well as run time. Initialization generally must be done in a specific sequence which will depend on the environment existing at that time. An abortive initialization environment may require an orderly shut-down of the system. The operating system has the capability for managing these situations, including communications, access to system resource information, and displaying status of the initialization actions.

Debugging

The system debugger is a window into the internal structures of the operating system. Debuggers allow data structures and memory to be examined, breakpoints to be set, and the user to be notified of abnormal conditions. The debugger may have symbolic debugging, in which system objects such as addresses, tasks, jobs, mailboxes, and memory locations can be assigned names. This gives greater flexibility and accuracy during debugging. The debugger may not be necessary in the final system, so the debugger is often a separate system job. This allows removal of the debugger with no effect on the remaining jobs.

Debugging will be aided if the operating system verifies the parameters required by system actions and also returns status for the requested action. Parameter verification is particularly valuable for new program code in a developing system. Status results other than success are abnormal or exception conditions. Exception conditions may include insufficient memory for the request, invalid input data, inoperative I/O devices, an invalid request for an action, or a request for an invalid action. The operating system may have an exception handler for these conditions and may allow the debugger to be used as the exception handler. The development process will be more efficient if detection of exception conditions takes place for all levels of system actions, from initialization of jobs and tasks to requests for memory or status.

Higher Level Functions

With the continuing increase in system complexity, more operating systems are providing higher level functions. These functions may include advanced I/O file management, operator console, spooling operations, telecommunications support, multiuser support and access to system resources of increasing size and complexity. Only the largest operating systems provide all of these capabilities, but users of component hardware must be careful their system will integrate higher level functions that may be required in future applications.

Extendibility

In order to provide general purpose support, operating systems must be extendible. New applications may require data structures or system actions not available with the present operating system. The system must be able to integrate these new structures and actions, supporting them in the same manner as existing functions. Choosing an operating system requires a large commitment, both in initial expense and system architecture. Extendibility provides assurance the operating system chosen will not provide built-in obsolescence of that commitment.

iRMX 86™ OPERATING SYSTEM ARCHITECTURE

Layers

The iRMX 86 Operating System architecture is shown in Figure 3. It includes the Nucleus, Basic I/O System, Extended I/O System, Applications Loader, and Human Interface. These major portions of the operating system are designed as layers. Each layer may be added to previous layers as application needs grow. Lower layers may be used without upper layers. All layers may reside in programmable read only memory. Applications have access to all portions of the system, from the Nucleus to all outer layers.

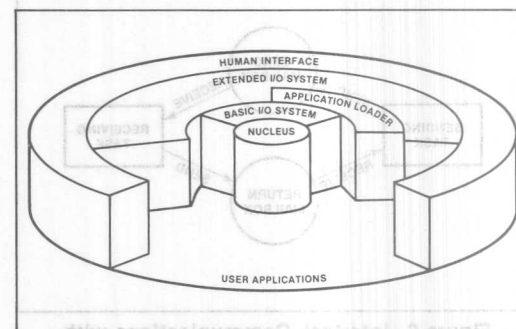


Figure 3. Architecture of the iRMX 86™ Operating System

The Nucleus is the heart of the system. It includes support for multiprogramming, multitasking, communications, synchronization, scheduling, resource management, extendibility, interrupt handling, and error detection. The Nucleus may be considered as an extended layer of the underlying hardware, giving the hardware system management functions and making the software independent of the detailed hardware. The system environment, including resources, priorities, and placement of program code, is made known to the Nucleus at system initialization. All requests for memory, communication, and creation of basic data structures must

go through the Nucleus. These requests are made by system calls, which are comparable to subroutine calls for system actions.

All higher level functions of the iRMX 86 Operating System are built around a core of the Nucleus. Although outer layers may require a substantial number of the system functions included in the Nucleus, the Nucleus itself is configurable on a call-by-call basis. "Configurable" means the Nucleus may be altered so it contains code only for those functions required by the application. Certain features, such as parameter validation and exception handling, are also configurable. Features and system calls may be included for development and excluded from the final system, giving a Nucleus tailored for each level of application development.

The Basic I/O System is the first layer above the Nucleus. The Basic I/O System provides asynchronous I/O support and format independent manipulation of data. Multiple file types are supported, including Stream, Named, and Physical files. Stream files are internal files for transferring large amounts of data between jobs or tasks. Named files include data files of varying sizes and directories for those files. Named files are designed for random access disk storage. Physical files consider the entire device to be one file. Physical files are primarily used to transfer data to and from printers, tape drivers, and terminals. Device drivers for both floppy and hard disks are provided. Like the Nucleus, the Basic I/O System provides system calls to invoke I/O actions. These calls and the features of the Basic I/O System are fully configurable.

The Application Loader provides the ability to load code and data from mass storage devices into system RAM memory. The Application Loader resides on the Basic I/O System, allowing application code to be loaded from any random access device supported by the Basic I/O System. Application code can be loaded and executed as needed rather than residing in dedicated system memory.

The Extended I/O System supports synchronous I/O, automatic buffering, and logical names. Synchronous I/O provides a simplified user interface for I/O actions. Automatic buffering improves I/O efficiency by overlapping I/O and application operations wherever possible. Logical name support allows applications to access files with a user-selected name, aiding I/O device independence.

The Human Interface uses all lower layers, forming a high level man-machine interface for user program invocation, command parsing, and file utilities. The primary purpose of the Human Interface is to support the addition of interactive commands. The Human Interface is the basis for pass-through language support and multiple user systems.

Configurability

Configurability means the iRMX 86 Operating System can be changed to include only the system calls and features pertinent to the system under development. Smaller applications start with only the iRMX 86 Nucleus. A subset of Nucleus calls, described later in this application note, provide the basis for management of jobs, tasks, memory, interrupts, communication and synchronization, and support for the Debugger and Terminal Handler.

All systems calls may also use parameter validation as a configuration option. Parameter validation verifies that system calls reference correct system objects before the requested action is performed. During debugging and in hostile environments, validation provides error detection for each system call. This error detection does add some overhead to the calls. Debugged application jobs can perform more efficiently without the validation, while new code can use parameter validation to speed development.

Once errors are detected, there are two means available to handle error recovery. The task can either use the status information to perform error recovery actions or the recovery actions may be performed by a specialized error handling program called an exception handler. Applications may use the Debugger as an exception handler, or use one implemented by the application. There are two classes of errors that may cause control to be given to an exception handler; avoidable errors, such as programmer errors, and unavoidable errors, such as insufficient memory. Exception handlers can be selected to receive control for either or both classes.

Support Functions

The iRMX 86 Operating System includes a Debugger, a Terminal Handler, a Bootstrap Loader, and a Patch Facility. The Debugger examines system objects, using execution and exchange (mailbox and semaphore) breakpoints, symbolic debugging, and exception handling. The Terminal Handler provides a line-editing, mailbox-driven CRT communications capability. The Bootstrap Loader is a fully configurable loader for bootstrap loading on reset or command, from any specific random access device. The Patch Facility gives the capability of patching iRMX 86 Object Code in the field.

Object Orientation

The iRMX 86 Operating System is based on a set of system data structures called objects. These objects include jobs, tasks, mailboxes, semaphores, segments, and regions. Users may also define application-specific objects. Object architecture includes the objects, their parameters, and the functions allowed with the objects.

Object orientation is a formal, hardware-independent definition of hardware-dependent system structures that are currently used by most applications. For example, without object orientation, memory is reserved in advance for system buffers. The application code knows buffer sizes and locations. If buffer requirements grow, requiring a new memory layout, much of the application code will change to accommodate the new buffer sizes and locations. Using object orientation, the application requests a segment (buffer) of a particular size when the buffer is needed. The Nucleus allocates the memory and returns a segment object to the application. If the application needs a larger buffer, it returns the old segment and requests a new one of a larger size. The application obtains more buffers by making requests for more segments. If the hardware changes, the iRMX 86 Operating System is made aware of the changes. The application code uses the same system calls to request and return the segment objects regardless of the hardware configuration.

Objects are provided for modular environments (jobs), application code functions (tasks), communication (mailboxes), synchronization (semaphores), memory (segments), and mutual exclusion (regions). Objects are fundamentally a set of standard interfaces between application code and the iRMX 86 Operating System. The standard interfaces have three primary benefits:

- 1) First, objects provide structures, such as tasks or segments, that are common to all applications. The structures form the basis for a standard set of system calls that make the interface between the application and the operating system more consistent and easier to learn. These calls allow applications to create more objects (segments for buffers, for example), delete them, change them, and inspect them. Development engineers can use their knowledge of the objects on many applications, rather than just the one under development. The common objects allow a common system debugger to be used. The debugger will work for all applications, letting engineers concentrate their efforts on the application itself rather than designing and implementing custom debugging tools.
- 2) Second, the standard characteristics of the object allow consistent error detection and handling. Requests to alter or use objects can be checked for validity before the Nucleus actually performs the request. Errors can be classed as common to all objects or specific to certain objects, giving more precise error information for effective error handling and faster debugging.
- 3) Third, the object interface will be preserved on future releases of the iRMX 86 Operating System. Current application code can be split into independent modules. Future applications can use the modules for

common functions, preserving the investment in application software.

Task Scheduling

The Nucleus controls task scheduling by task priority and task state. Task priority is specified when the task is created. The priority can be altered dynamically. Tasks are classified into one of five classes: Running, Ready, Asleep, Suspended, or Asleep-Suspended. Tasks that have not been created are considered to be non-existent. The State Transition Diagram is shown in Figure 4.

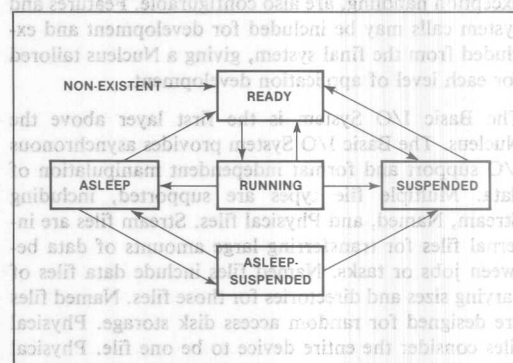


Figure 4. State Transition Diagram

Only one task is in the Running state. This task has control of the central processor. The Ready state is occupied by those tasks that are ready to run but have lower priority than the Running task. The Asleep state is occupied by tasks waiting for a message, semaphore units, availability of a requested resource, an interrupt, or for a requested amount of time to elapse. A task can specify the amount of time it will allow itself to spend in the Asleep state, but tasks in the Suspended state must be "resumed" by other tasks. The Suspended state is useful when situations require firm scheduling beyond the control provided by priority and system resource availability. Examples of these situations are system emergencies, controlling tasks in the Ready state for application-dependent scheduling algorithms, and guaranteeing a fixed initialization or shut-down sequence. If another task "suspends" a task already in the Asleep state, the sleeping task goes to the Asleep-Suspended state. This task will enter the Suspended state if the sleep-causing condition is satisfied. The task will go to the Asleep state from the Asleep-Suspended state if it is resumed before the sleep-causing condition is removed. If a task enters the Ready state and has higher priority than the present Running task, the Ready task is given control of the CPU. Control is transferred to another task only when:

- 1) The Running task makes a request that cannot immediately be filled. The Running task is moved to the

- Asleep state. The highest-priority Ready task becomes the Running task.
- 2) An interrupt occurs, causing a higher-priority task to become Ready. The current Running task goes to the Ready state, allowing the higher-priority task to become the Running task.
 - 3) The Running task causes a higher-priority task to become Ready by releasing the resource for which the higher-priority task is waiting. The current Running task goes to the Ready state. The higher-priority task becomes the Running task.
 - 4) The Running task causes a higher-priority task to become Ready by sending a message or semaphore units to the mailbox or semaphore where the higher-priority task is waiting. The Running task is moved to the Ready state. The higher-priority task becomes the new Running task.
 - 5) The Running task removes a higher-priority task from the Suspended state by "resuming" it, placing the higher-priority task in the Ready state. The current Running task is moved to the Ready state and the higher-priority Ready task becomes the new Running task.
 - 6) The Running task creates a higher-priority task. The new task goes to the Ready state. The current Running task is moved to the Ready state and the higher-priority Ready task becomes the new Running task.
 - 7) The Running task places itself in the Suspended state. The highest-priority Ready task becomes the new Running task.
 - 8) The Running task places itself in the Asleep state. The highest-priority Ready task becomes the new Running task.
 - 9) The Running task deletes itself, becoming Non-existent. The highest-priority Ready task will be the new Running task.

System Hardware Requirements

The iRMX 86 Operating System will run on any system that meets the following minimum hardware requirements:

- 1) An iAPX 86 or iAPX 88 Central Processing Unit.
- 2) An Intel 8253 Programmable Interval Timer to provide the system clock.
- 3) An Intel 8259A Programmable Interrupt Controller.
- 4) Enough hardware to provide a system clock and bus interfaces. This may be supplied by the Intel 8284 Clock Generator, Intel 8288 Bus Controller, Intel 8282/8283 Latches, and Intel 8286/8287 Transceivers.

- 5) The following RAM:

- a. 1024 bytes from 0 to 1024 for software interrupt pointers (the interrupt vector).
- b. 800 bytes for Nucleus data.
- c. Enough RAM for the application data, code, and system objects.

- 6) Enough EPROM or RAM to hold the required parts of the iRMX 86 Operating System and the application code.

The Intel iSBC 86/12A Single Board Computer more than meets these minimum requirements. A block diagram of the board is shown in Figure 5. Note in addition to the timer and interrupt controller the board contains an 8251A USART, an 8255 parallel I/O interface, a MULTIBUS™ interface, four sockets for up to 16K bytes of EPROM, and 32K bytes of dual-ported RAM. Even though a user may be developing a custom board for his application, it is recommended that initial system development be accomplished using the iSBC 86/12A Single Board Computer. This will provide a known hardware environment to simplify debugging. In addition, the development hardware system can be adapted to changing application needs by adding Intel MULTIBUS compatible boards to the iSBC 86/12A Single Board Computer. After the software is fully debugged, the application can be moved to the final custom hardware design.

APPLICATION EXAMPLE

A spectrum analyzer is the subject of this application. The analyzer displays the frequency spectrum of an analog signal on a general purpose CRT terminal. The user has control over input signal bandwidth, averaging, and continuous analysis. A fast Fourier transform (FFT) program is used to obtain frequency data from samples of analog data. Fourier transforms provide useful frequency analysis, but the large processing requirements of Fourier transforms have restricted their use. Fast Fourier transforms take advantage of the repetitive nature of the Fourier calculations, allowing the Fourier transforms to be completed significantly faster. The FFT used in this application note is known as "time decomposition with input bit reversal."¹ Sixteen-bit integer samples of the input signal are placed in frames, with each frame holding 128 complex points. An averaged power spectrum is calculated to sum and square the FFT values, yielding 64 32-bit power spectrum values. These values are displayed on a standard CRT terminal.

1. S. O. Stearns, *Digital Signal Analysis*, Hayden Book Co., Rochelle Park, NJ, 1975.

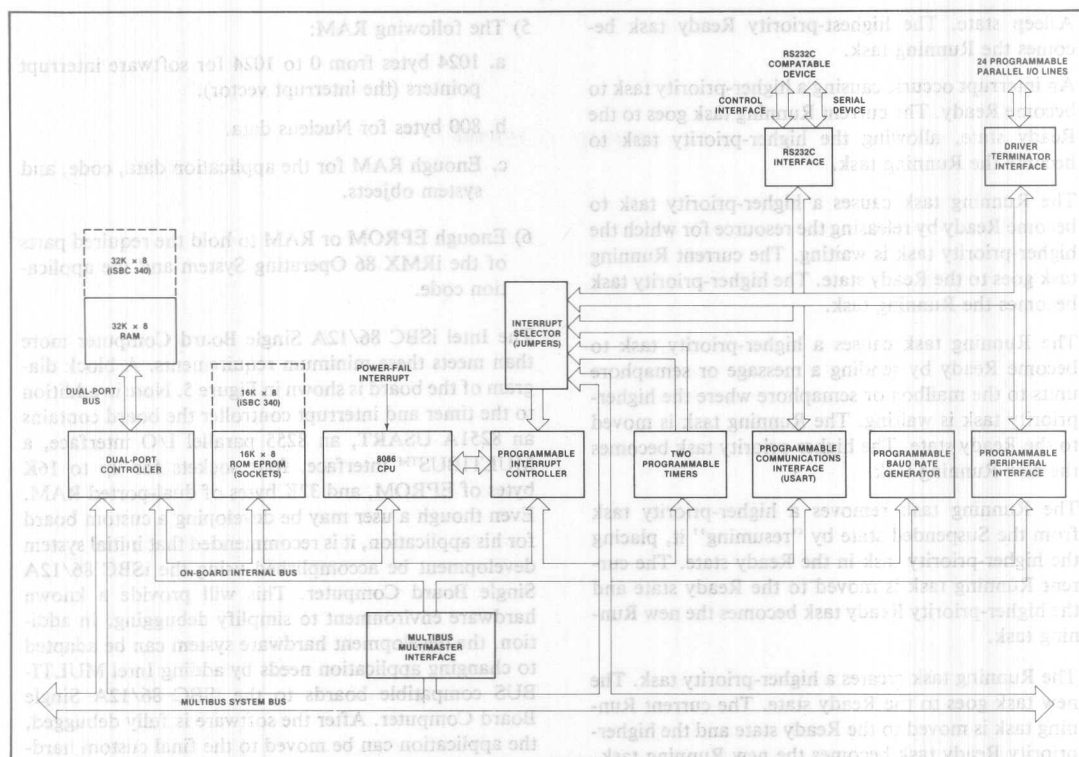


Figure 5. iSBC 86/12A™ Single Board Computer Block Diagram

The FFT algorithm may be applied wherever frequency analysis of an analog signal is required. Medical applications for FFTs include EEG analysis, blood flow analysis, and analysis of other low-frequency body signals. Industrial uses are production line testing, wear analysis, frequency signature monitoring, analysis in noisy or hostile environments, and vibration analysis. Other applications could cover remote reduction of analog data, frequency correlation, and process control. For this application note, the actual use of the FFT is secondary to its existence as a modular, CPU-intensive task in a general purpose system.

The overall application system characteristics are the following:

- 1) A user-selectable input signal bandwidth of 120 Hz, 600 Hz, 1200 Hz, 6000 Hz, or 12,000 Hz.
- 2) The option of averaging frames of samples. The averaging is user selectable, with options of 1 (no averaging), 2, 4, 8, 16, or 32 frames averaged per CRT display.
- 3) The capability, also user selectable, of repeating the analysis cycle continuously.

- 4) User capability to abort the analysis.
- 5) Twelve-bit input sample resolution.
- 6) A minimum of hardware requirements, including no more than 32K bytes of EPROM memory and 16K bytes of RAM memory.
- 7) A standard character screen CRT for output.
- 8) A multitasking structured design that will use a subset of the iRMX 86 Nucleus and exhibit modular application code, formal interfaces, and self-documentation.

System Design

To begin the design, the application is broken up into functional modules, much the same as a hardware block diagram. A SUPERVISOR TASK initializes the system, accepts operator parameters, starts the analysis cycle, and stops the processing upon cycle completion or operator request. An INPUT TASK samples the data and places it in a buffer. An FFT TASK receives the buffer and processes the data. An OUTPUT TASK displays the data received from the FFT TASK. This structure is shown in Figure 6.

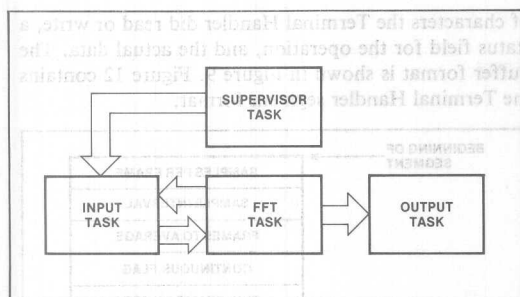


Figure 6. Basic Application Architecture

The general task functions are:

SUPERVISOR TASK: The SUPERVISOR TASK initializes the system by creating the other tasks. The SUPERVISOR TASK then obtains the analysis parameters from the operator. Each parameter is verified as it is received. When all of the parameters are accepted, the operator is asked if they are satisfactory. If the operator agrees, the SUPERVISOR TASK sends frame buffers to the INPUT TASK to initialize the analysis. If not, the operator is asked to input all of the parameters again. During the FFT analysis, the SUPERVISOR TASK waits for an abort request from the operator and for the frame buffer from the OUTPUT TASK. If the abort request is received, the SUPERVISOR TASK terminates the analysis in an orderly fashion and asks the operator for parameters for the next analysis cycle. If the frame buffer is received from the OUTPUT TASK and continuous analysis has been selected, the SUPERVISOR TASK sends the frame buffer to the INPUT TASK to start the next cycle. If the frame buffer is received from the OUTPUT TASK and continuous analysis has not been selected, the current analysis is complete and the SUPERVISOR TASK asks for new parameters.

INPUT TASK: The INPUT TASK receives the frame buffer from the SUPERVISOR TASK. The input signal is sampled according to the analysis parameters. The actual sample rates are calculated as follows:

- 1) Multiply the highest frequency of interest by two to obtain the Nyquist sampling rate.
- 2) Invert this value to obtain time between samples.
- 3) Scale the value by 60/64. The CRT display is limited to 64 columns. The scaling maps sample values to columns 1 to 60 rather than 1 to 64, giving a more readable x-axis label and display. This method yields sample times of 3.9 milliseconds, 781 microseconds, 390 microseconds, 78 microseconds, and 39 microseconds for frequencies of 120 Hz, 600 Hz, 1200 Hz, 6000 Hz, and 12,000 Hz.

The INPUT TASK samples the data at the required interval and places the samples in the frame buffer. When the frame buffer is full, the INPUT TASK up-

dates the frame buffer number and sends the frame buffer to the FFT TASK. The INPUT TASK sends a status message to the CRT terminal and waits for the next frame buffer.

FFT TASK: The FFT TASK receives the frame buffer from the INPUT TASK. A fast Fourier transform is performed on the data contained in the buffer, the power spectrum is calculated, and the data is averaged with previous data if necessary. If the frame buffer is the last one to be averaged prior to display of the frequency data, the frame buffer is filled with the averaged data and sent to the OUTPUT TASK. If the buffer is not the last one to be averaged, the FFT TASK returns the buffer to the INPUT TASK for another frame of data or to the SUPERVISOR TASK if the analysis cycle is nearly complete. The FFT TASK sends status information to the CRT display and waits for the next frame buffer from the INPUT TASK.

OUTPUT TASK: The OUTPUT TASK receives the frame buffer from the FFT TASK. The data is formatted and displayed. The OUTPUT TASK sends the frame buffer to the SUPERVISOR TASK and waits for the next frame buffer from the FFT TASK.

Terminal Handler: The Terminal Handler serves as the basic I/O device for parameter requests, status data, and frequency displays. The Terminal Handler accepts display requests from all tasks and sends operator input to the SUPERVISOR TASK.

The basic functions of the various tasks in the application have been defined, but system integration has not been discussed. Synchronization of the tasks, scheduling, resource management, mapping to hardware, interrupt handling, and system interfaces have been omitted. No debugging functions have been defined. It is clear the system implementation is just started. The iRMX 86 Nucleus will provide all of the system integration "glue" the application requires, allowing application programmers to concentrate on the actual functional code. In order to use this "glue," the application must be divided into jobs and tasks.

Jobs and Tasks

The iRMX 86 Operating System architecture defines jobs as separate environments within which tasks operate. These separate environments allow each job to function with no knowledge of other system jobs. There are two jobs in this application, the Debugger job and the Application job.

The jobs contain functional portions or working programs called tasks. The Application job contains the INIT TASK, SUPERVISOR TASK, INPUT TASK, FFT TASK, OUTPUT TASK, and INTERRUPT TASK. The Debugger job contains the Debugger task and the Terminal Handler task. Tasks provide the ap-

plication goals of modularity, resource constraint boundaries, and functional independence. The structure of the development system is shown in Figure 7.

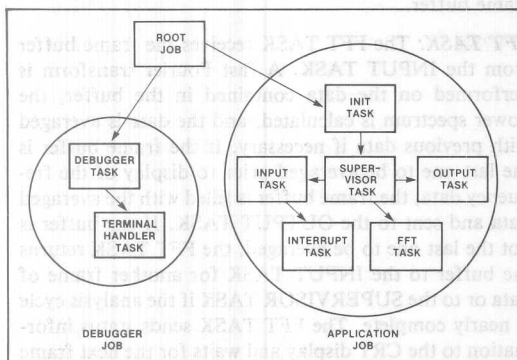


Figure 7. Development System Job Structure

The Debugger job is included only for development. When development is completed, the Debugger job is removed from the system. A Terminal Handler job containing only the Terminal Handler task is substituted in place of the Debugger job. The application code is not changed. This structure is shown in Figure 8.

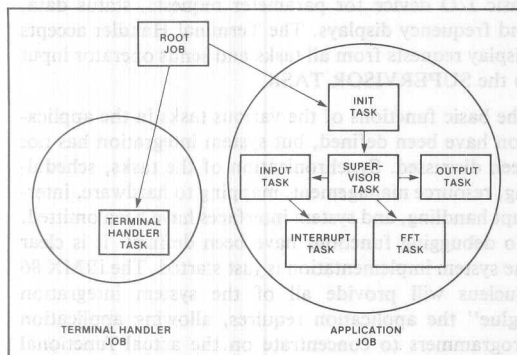


Figure 8. Final System Job Structure

Interfaces and Synchronization

Now that the system is made of jobs and tasks, the primary need is to synchronize the tasks and provide communication interfaces. This will be handled by mailboxes. The messages sent via the mailboxes will be segments, which are pieces of memory allocated by the Nucleus. The frame buffers sent from task to task are these segments. The buffer segments contain all the analysis parameters in addition to the data samples. Communication with the Terminal Handler task is also accomplished by mailboxes, but with a different buffer format. The Terminal Handler format has fields for the operation requested (read or write), the number of characters the task wishes to read or write, the number

of characters the Terminal Handler did read or write, a status field for the operation, and the actual data. The buffer format is shown in Figure 9. Figure 12 contains the Terminal Handler segment format.

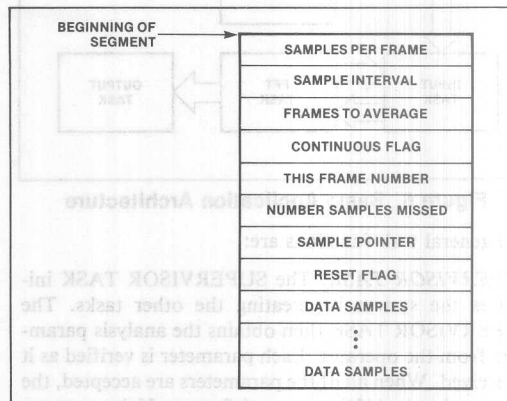


Figure 9. Frame Buffer Format

Mailboxes are used to pass the buffer segment from task to task. Tasks can send segments to mailboxes, or receive segments from mailboxes. If there is no segment at a mailbox, a task can elect to wait for the segment, with the wait duration ranging from zero to forever. This provides the simplest system synchronization — each task, upon initialization, waits at its input mailbox for a frame buffer segment. When the task receives the segment, it processes the data, sends the segment on to the mailbox for the next task, and returns to its own mailbox to wait for the next frame buffer. The system is synchronized and controlled by the availability of frame buffers and task priority. It should be clear that multiple frame buffers segments provide overlapped processing, with the segments ultimately “piling up” at the slowest task in the chain. This loose coupling arrangement allows tasks to have radically different execution times. For example, the INPUT TASK has an input sampling time that ranges from 0.6 seconds to 6.3 milliseconds, a range of 100 to 1, and the system requires no special synchronization or scheduling to accommodate this range.

The mailbox interfaces, shown in Figure 10, serve several other important purposes. First, they provide the standardized interface that is a goal of this application. The set of mailboxes and the two buffer segments form all inter-task interfaces. Each task uses only a few mailboxes, making it easy to add or remove tasks by adding or removing mailboxes. The system could easily be expanded to include data reduction tasks, data correlation tasks, or to substitute different tasks for any of the present ones. Dummy tasks were used for real tasks during development to verify overall system execution before the actual tasks were available.

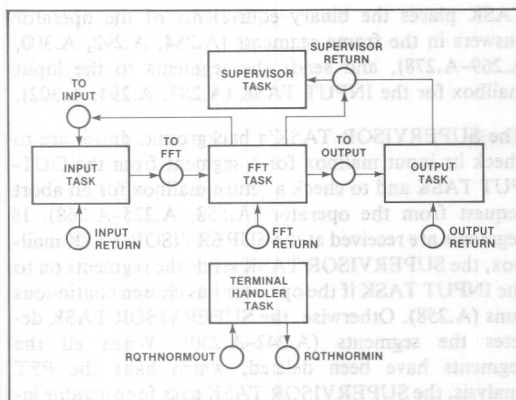


Figure 10. Application Architecture with Mailboxes.

The mailboxes also provide a very convenient window into the application system processing for both debugging and aborting the current cycle. The Debugger can set breakpoints at mailboxes to allow users to examine the frame buffers as they progress from task to task. The Debugger can examine buffer data and control the processing cycle. Tasks wait at the mailboxes in a queue that is either priority or first-in-first-out (FIFO) based. The inter-task mailbox queues are priority based, which means the higher priority SUPERVISOR TASK can intercept segments at the mailboxes ahead of the lower priority waiting tasks, and abort the analysis by removing all of the buffer segments. This method of aborting requires no knowledge of the internal processing of the tasks, making it universally applicable to all the tasks.

A return mailbox may be specified when a segment is sent to a mailbox. The receiving task may send status information, a different segment, or the original segment to the return mailbox. The Terminal Handler will return

the buffer segment sent to it if a return mailbox is specified. This is used to synchronize the tasks with the Terminal Handler and to allow multiple tasks to use a single display task. Each sending task waits at a separate return mailbox for the Terminal Handler to return its segment. Each task retains control over its buffer segment and is synchronized with the slower data display function.

Priority

In addition to the mailboxes, task execution is governed by priority. In this system, the INPUT TASK has maximum priority to guarantee it can sample the input signal at the precise intervals required for the FFT. The SUPERVISOR TASK, responsible for abort functions, has the next level of priority, with the FFT TASK next, and the OUTPUT TASK lowest.

APPLICATION IMPLEMENTATION

Display Functions

All application tasks use the iRMX 86 Terminal Handler as an output device. The Terminal Handler is chosen because it provides a standard interface consistent with the application goals, it exists both in the Debugger job and in an independent job, and it is easy to integrate into the application system. The application could also use the same interface with a user-written Terminal Handler. In this application, the Terminal Handler can have messages from four tasks on the screen at one time. To allow this to occur in an orderly fashion, lines on the screens are reserved for each of the tasks. The screen format is shown in Figure 11. Each message to the screen sends the cursor to the upper left corner (the home position), then down to the proper line to display the data.

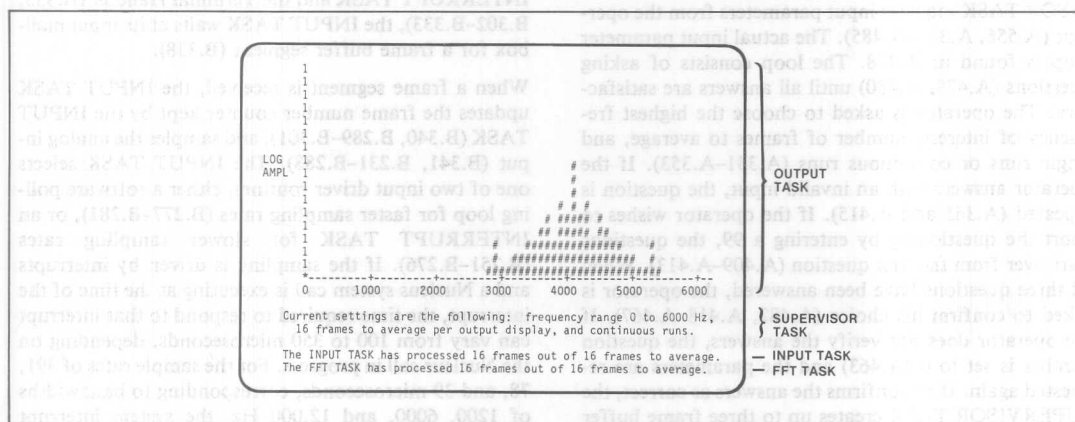


Figure 11. CRT Screen Display Format

Cataloging

To aid the debugging process, all system objects, such as mailboxes, segments, and tasks, are cataloged in the directory of the SUPERVISOR TASK. The catalog entries are user-selected, 12-character names. The Debugger can display this directory, giving easy access to objects to aid symbolic debugging. If other tasks know the proper directory and the 12-character name, the tasks can look up the objects in the directory and obtain access to them. This is the method used to find the Terminal Handler mailboxes. For objects that are cataloged only to aid debugging, the system calls that catalog the objects are removed from the code when debugging is complete.

Application Code

The code listings for the SUPERVISOR TASK and the INPUT TASK are included in appendices to this note. Code listings for other tasks are not included, but they are available from the Intel Insite software library. The following discussions reference line numbers in the listings included in this note. The references begin with a first letter for the appendix section (A or B), followed by the actual line number (A.220, for example).

SUPERVISOR TASK

Code listings for the SUPERVISOR TASK are in Appendix A. The actual SUPERVISOR TASK procedure begins at line A.550. After initializing internal buffers and mailboxes (A.551, A.518–A.549), the Supervisor sends an initial screen, one line at a time, to the Terminal Handler (A.553, A.502–A.517). When the screen is complete, the SUPERVISOR TASK creates the INPUT TASK, FFT TASK, and OUTPUT TASK (A.554, A.486–A.501). The order of creation is not important for this application, as each task begins by waiting at its input mailbox for frame buffer segments. The SUPERVISOR TASK requests input parameters from the operator (A.556, A.305–A.485). The actual input parameter loop is found at A.478. The loop consists of asking questions (A.479, A.480) until all answers are satisfactory. The operator is asked to choose the highest frequency of interest, number of frames to average, and single runs or continuous runs (A.331–A.353). If the operator answers with an invalid input, the question is repeated (A.365 and A.415). If the operator wishes to abort the questioning by entering a 99, the questions start over from the first question (A.409–A.413). When all three questions have been answered, the operator is asked to confirm his choice (A.482, A.418–A.467). If the operator does not verify the answers, the question number is set to 0 (A.463) and the parameters are requested again. If he confirms the answers as correct, the SUPERVISOR TASK creates up to three frame buffer segments (A.557, A.279–A.304). The SUPERVISOR

TASK places the binary equivalents of the operator answers in the frame segments (A.284, A.292, A.300, A.269–A.278), and sends the segments to the input mailbox for the INPUT TASK (A.287, A.294, A.302).

The SUPERVISOR TASK's background duties are to check its input mailbox for a segment from the OUTPUT TASK and to check a return mailbox for an abort request from the operator (A.558, A.225–A.268). If segments are received at the SUPERVISOR TASK mailbox, the SUPERVISOR TASK sends the segments on to the INPUT TASK if the operator has chosen continuous runs (A.258). Otherwise, the SUPERVISOR TASK deletes the segments (A.242–A.250). When all the segments have been deleted, which halts the FFT analysis, the SUPERVISOR TASK asks for operator input again (A.556).

If an operator abort request is received, the SUPERVISOR TASK, having higher priority than the FFT or OUTPUT TASKS, waits at their mailboxes to intercept the frame segments (A.243, A.249, A.207–A.224). When a segment is received it is deleted (A.219). The SUPERVISOR TASK also checks the INPUT TASK mailbox under abort conditions (A.244). This mailbox is FIFO based to allow the SUPERVISOR TASK to intercept the buffer segment ahead of the higher-priority INPUT TASK (A.523). The SUPERVISOR TASK input mailbox is also checked for frame buffer segments that may have been sent there by the OUTPUT TASK after the abort was requested (A.246). When all of the frame buffer segments have been deleted, the SUPERVISOR TASK asks for operator input (A.556).

INPUT TASK

Listings of the INPUT TASK are in Appendix B. After initializing the buffer for Terminal Handler communications and the mailboxes for communicating with the INTERRUPT TASK and the Terminal Handler (B.335, B.302–B.333), the INPUT TASK waits at its input mailbox for a frame buffer segment (B.338).

When a frame segment is received, the INPUT TASK updates the frame number counter kept by the INPUT TASK (B.340, B.289–B.301), and samples the analog input (B.341, B.231–B.288). The INPUT TASK selects one of two input driver routines, either a software polling loop for faster sampling rates (B.277–B.281), or an INTERRUPT TASK for slower sampling rates (B.251–B.276). If the sampling is driven by interrupts and a Nucleus system call is executing at the time of the interrupt, the time required to respond to that interrupt can vary from 100 to 350 microseconds, depending on the Nucleus call in progress. For the sample rates of 391, 78, and 39 microseconds, corresponding to bandwidths of 1200, 6000, and 12,000 Hz, the system interrupt latency cannot guarantee the precise sampling interval

required. A simple software polling loop with a delay between samples is used for these rates (assembler code for this loop is included after the INPUT TASK listings in Appendix B). This loop operates at priority 0, the highest priority, to guarantee the loop is not interrupted (B.278, B.280) while the sampling is in progress.

For the longer intervals of 3.9 milliseconds and 781 microseconds, corresponding to bandwidths of 120 and 600 Hz, an Interrupt Handler and an INTERRUPT TASK are used (B.251–B.276). Under the iRMX 86 Operating System architecture, an Interrupt Handler is defined as a short procedure with a primary goal of fast interrupt response and limited Nucleus calls. All hardware interrupt levels are masked when an Interrupt Handler is responding to an interrupt. If the interrupt servicing requires higher-level system functions, the Interrupt Handler notifies a waiting INTERRUPT TASK. Higher-level interrupts are enabled when an INTERRUPT TASK is executing. INTERRUPT TASKS can make all system calls.

The INTERRUPT TASK (B.196–B.203) binds the Interrupt Handler to the hardware interrupt level (B.197) and waits for a signal from the Interrupt Handler (B.199). The Interrupt Handler (B.164–B.195) verifies the interval accuracy (B.166–B.173), samples the data (which automatically starts the next sample) (B.175–B.176), places the data in the frame buffer (B.181–B.184), and notifies the INTERRUPT TASK when the frame buffer is full (B.193). If the buffer is not full, the Interrupt Handler resets the interrupt hardware (B.194). The INTERRUPT TASK notifies the INPUT TASK (B.200) and waits for a return message (B.201). The INPUT TASK disables interrupt level 3 (B.274) and returns the token to the INTERRUPT TASK (B.275). The INTERRUPT TASK enables the Interrupt Handler (B.199), but no interrupts will be received from the free-running 8253 Timer because hardware interrupt level 3 has been disabled. Sampling for the next buffer is initiated by simply enabling level 3 (B.272). The INPUT TASK sends a status message to the Terminal Handler (B.342, B.219–B.230) and sends the filled frame buffer to the FFT TASK (B.343). The INPUT TASK then returns to the INPUT TASK input mailbox to wait for the next frame segment (B.338).

FFT TASK

Listings for the FFT TASK are not included with this application note. The FFT TASK is similar in overall format to the INPUT TASK. The FFT TASK waits at its input mailbox for a frame buffer segment from the INPUT TASK. When one is received, the FFT TASK computes the fast Fourier transform of the data. The auto power spectrum is computed and averaged with previous data. The FFT TASK sends its status message to the Terminal Handler for display. If the frame buffer

is the final one to be averaged, the FFT TASK sends the frame buffer to the OUTPUT TASK. If the frame buffer is not the final one in this averaging series, the FFT TASK checks to see if the sampling process is continuous. If so, the frame buffer is returned to the INPUT TASK. If the sampling process is not continuous and the buffer is within two frames of the final frame buffer, the buffer is returned to the SUPERVISOR TASK to prevent unnecessary buffers from going to the INPUT TASK. The FFT TASK then returns to its input mailbox to wait for the next frame buffer.

OUTPUT TASK

Listings for the OUTPUT TASK are not included in this application note. The OUTPUT TASK, like the other tasks, waits at its input mailbox for a buffer. When a frame buffer is received from the FFT TASK, the OUTPUT TASK stores the data in an internal buffer and sends the frame buffer to the SUPERVISOR TASK.

The OUTPUT TASK converts each 32-bit frame buffer value to one of 16 levels by taking the base 2 logarithm of the significant 16 bits of sample value. The display screen is sent to the Terminal Handler one line at a time. Each line of the display is composed of 7 characters of label and y-axis data and 64 characters of display data (reference Figure 11). Each line of the display represents a power of two (from 16 down to 1). The character to be displayed at each location is found by comparing the appropriate sample value against the current line value. If the sample value is greater than the line number, a pound sign is displayed at that location. Otherwise, a space is displayed. The x-axis and labels are sent after the data lines to complete the display. The OUTPUT TASK then waits at its input mailbox for another frame buffer.

TERMINAL HANDLER

The Terminal Handler interfaces to the application tasks via the two mailboxes and buffer segment format shown in Figure 12. If a task wishes to display data, a segment containing the data is sent to the RQ\$TH\$NORM\$OUT mailbox, specifying a return mailbox. The Terminal Handler displays the data, updates the status fields, and sends the segment to the return mailbox.

Input proceeds in much the same fashion. A task requesting data sends a segment to the RQ\$TH\$NORM\$IN mailbox, again specifying a return mailbox. When the operator terminates the input line with a carriage return, the Terminal Handler puts the data in the segment, updates the status fields, and sends the segment to the return mailbox. This serves two primary purposes: specifying return mailboxes allows multiple tasks to share the display screen while retaining

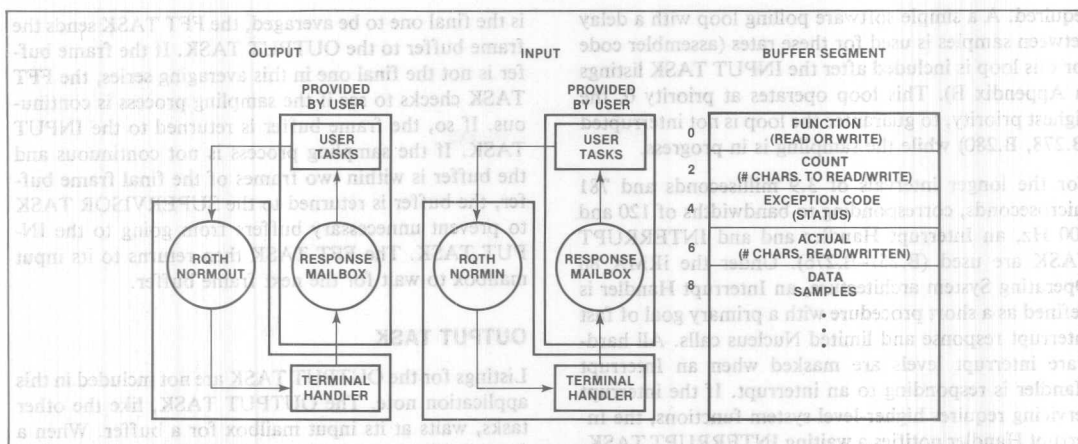


Figure 12. Terminal Handler Interface

synchronization and control over their data buffers; and a user-written Terminal Handler using the same protocol and mailbox names could easily be integrated into the application. For this application, the INPUT TASK, FFT TASK, OUTPUT TASK, and SUPERVISOR TASK all share the screen for output, but only the SUPERVISOR TASK uses the Terminal Handler to obtain operator input.

Nucleus Calls

The iRMX 86 Nucleus provides a comprehensive set of 61 system calls. A complete description of these calls may be found in the iRMX 86 Nucleus Reference Manual. For most applications, only a subset of the 61 calls will be required. The iRMX 86 Nucleus is configurable; which means the final system Nucleus will contain code only for the system calls required for the application. In this case, the following system calls were required:

RQ\$CREATE\$MAILBOX, RQ\$SEND\$MESSAGE, and RQ\$RECEIVE\$MESSAGE provide mailbox management.

RQ\$CREATE\$SEGMENT and RQ\$DELETE\$SEGMENT are used to create and delete segments for the frame buffers, internal buffers, and Terminal Handler. RQ\$SET\$INTERRUPT, RQ\$EXIT\$INTERRUPT, RQ\$SIGNAL\$INTERRUPT, RQ\$WAIT\$INTERRUPT, RQ\$ENABLE, and RQ\$DISABLE allow the INPUT TASK to handle hardware interrupts knowing only the hardware interrupt level (3).

RQ\$CREATE\$JOB, RQ\$CREATE\$TASK, and RQ\$SET\$PRIORITY are used to create the jobs and tasks, and set the priority of the input polling loop.

RQ\$GET\$TASK\$TOKENS, RQ\$LOOKUP\$OBJECT, RQ\$CATALOG\$OBJECT, RQ\$DISABLE\$DELE-

TION, RQ\$ENABLE\$DELETION, RQ\$GET\$TYPE, RQ\$GET\$PRIORITY, RQ\$GET\$SIZE, and RQ\$SIGNAL\$EXCEPTION are system calls required by the Debugger and the Terminal Handler. None of these calls are necessary in this application if a user-written Terminal Handler is used and debugging is completed.

System Configuration

System Configuration is the integration step in the development process. It consists of selecting the portions of the iRMX 86 Operating System required in the application, mapping this code and the application code to system memory, and creating a Root Job that will initialize the system. The overall configuration process is shown in Figure 13. Configuration requires knowledge of available memory, operating system and application code entry points, priorities, exception handlers, and other system parameters. System Configuration consists of the following steps:

- 1) Selecting the portions of the iRMX 86 Operating System required by the application, including the layers and the specific system calls in each layer.
- 2) Linking and locating those portions.
- 3) Assembling or compiling, linking, and locating the application code.
- 4) Creating a configuration file that will tell the Nucleus the locations of available RAM memory, initial characteristics of each system job, and pertinent overall system parameters. Each job in the system has an entry in the configuration file. The order of the entries is the order of initialization of the jobs.
- 5) Creating the Root Job by assembling, linking, and locating the configuration file.

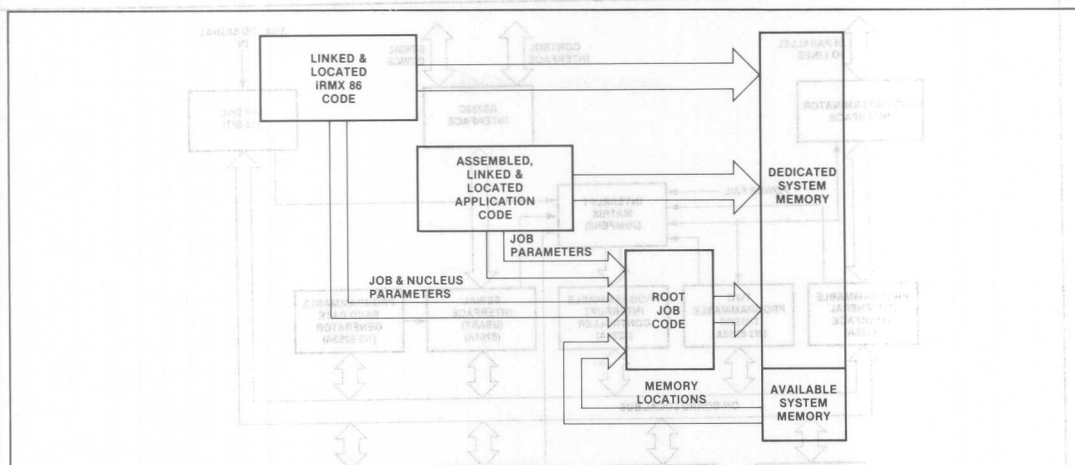


Figure 13. System Configuration Process

During development of an EPROM-based application such as this one, configuration is accomplished twice: once for the RAM-based development system and once for the final EPROM-based system. These configurations are detailed in Appendix C, System Configuration. In both cases, the Root Job that results from configuration initializes the system jobs. For development, the system job structure is shown in Figure 7. The Root Job creates the Debugger Task in the Debugger Job, which in turn creates the Terminal Handler Task. The Root Job then creates the SUPERVISOR TASK, which creates the INPUT TASK, FFT TASK, and OUTPUT TASK. The INPUT TASK creates the INTERRUPT TASK when necessary.

Software development is completed on the iSBC 86/12A Single Board Computer discussed earlier in this note. After application code is debugged and ready to be placed in EPROM memory, the Debugger Job, which contains both the Debugger and the Terminal Handler, is removed and replaced with the Terminal Handler Job, which contains only the Terminal Handler. This job structure is shown in Figure 8. The Nucleus system calls required only by the Debugger are removed from the iRMX 86 Nucleus. The application code is not changed. The application code and the iRMX 86 Nucleus is configured for the final system, put in EPROM memory, and tested on the final hardware system. The final Nucleus and application code required 30.5K bytes of EPROM, allowing room for future code changes and some expansion within the 32K system limit.

The final application hardware is shown in Figure 14. This system contains an iAPX 86/10 CPU, an 8259A Programmable Interrupt Controller, and an 8253 Pro-

grammable Timer. The three chips form the primary hardware requirements for the iRMX 86 Operating System. The system is assembled from Intel components, using standard support circuits and system schematics described in Intel documentation. The analog sampling circuitry is a 12-bit analog to digital converter (ADC) and a sample/hold circuit. Both the sample/hold circuit and the ADC are driven from the on-board local bus. The ADC has a conversion time of 35 microseconds, limiting the overall cycle to approximately 39 microseconds per sample, or a maximum CRT display bandwidth of 12,000 hz.

The hardware system shown in Figure 14 contains components not specifically required for the final configuration. The 8255A Programmable Peripheral Interface and the MULTIBUS multimaster interface are not necessary for a system limited to just spectrum analysis and display via the CRT. However, the flexibility advantages of the iRMX 86 Operating System are supported by this hardware. For instance, the frequency spectrum display is limited by the CRT to a 16-level logarithmic approximation. Accuracy could be improved by using the programmable peripheral interface to drive a plotter or an analog CRT via a digital to analog converter. Software drivers for the plotter or CRT could be new tasks, interfacing to the old tasks through the mailboxes. Or, the OUTPUT TASK could be simply replaced with a new OUTPUT TASK for the plotter and analog CRT. The inclusion of the MULTIBUS interface allows this application to be integrated into a larger system of MULTIBUS-compatible boards. MULTIBUS-compatible memory boards will also aid test and debug. Users of hardware components can include these modular Intel interfaces as required by their application, giving growth and configurability in both hardware and software.

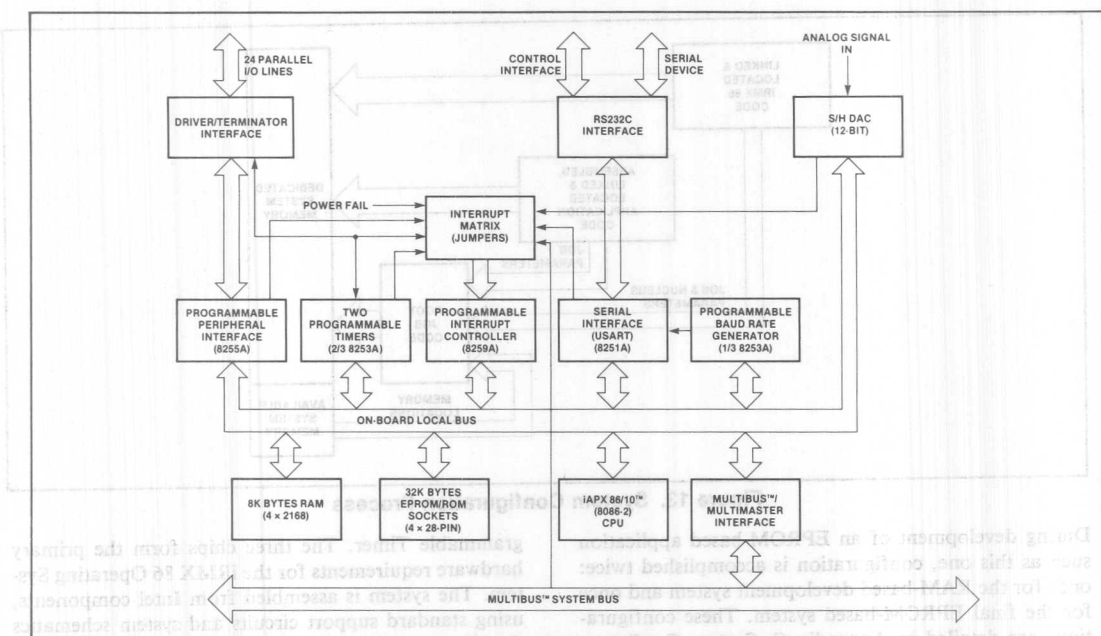


Figure 14. Final Hardware System Block Diagram

SUMMARY

This application note discussed general operating system functions, the Intel iRMX 86 Operating System, using the iRMX 86 Operating System on hardware component systems, and an example of an application implemented in a component environment. Users of the iRMX 86 Operating System are able to simplify application code development through modularity, standard interfaces, freedom from rigid hardware restrictions, and advanced debugging techniques. The iRMX 86 Operating System can be applied to larger systems by adding other iRMX 86 layers, making the software investment beneficial over a wide range of applications.

Operating systems provide many advantages for hardware component designs, but all of these benefits can be utilized only if the operating system and the development environment are fully supported. Intel's support for this application begins with an Intel MDS 230 Series II Microcomputer Development System. The MDS 230 System interfaces with the development hardware through the iSBC 957A™ Monitor. The development hardware is an Intel iSBC 86/12A Single Board Computer, an Intel iSBC 711™ Analog Input Board, an Intel iSBC 032™ 32K RAM Memory Board, an Intel iSBC 064™ 64K RAM Memory Board, and an Intel iSBC 660™ System Chassis. Final application hardware is debugged using Intel's ICE-86™ 8086 In-Circuit Emulator. Software support is provided by the ISIS-II

PL/M 86™ Compiler, MCS-86™ Macro Assembler, and the MCS-86 Utilities LINK86, LOC86, and OH86. The Intel UPP-103™ Universal PROM Programmer is used to convert the final system to PROM memory. This broad support allows expedient development of prototype and final systems based on the iRMX 86 Operating System.

The iRMX 86 Operating Systems and the Intel development tools are valuable only if they translate directly to increased productivity and shortened time to market for new products. This application has 1567 lines of application code. It was developed, from design to final implementation, in approximately 9 man-weeks of effort. This high level of productivity was achieved with the added benefits of modularity, standardization, and ease of application growth.

Intel Corporation is committed to both the continued integration of higher-level functions into hardware and to maintaining compatibility of present software with new hardware. One result of these commitments will be the Intel iAPX 86 and iAPX 286 Processors, which will be compatible with the iRMX 86 Operating System. Another result will be the placement of the iRMX 86 Operating System Nucleus into hardware. This will allow custom hardware applications to have higher-level functions, simplified development, and decreased chip count. Using the iRMX 86 Operating System today will give hardware component users a headstart on Intel's technological innovation for tomorrow.

PL/M-86 COMPILER SUPERVISOR TASK FOR AP NOTE 110, OCTOBER 1980
 ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE SUPERVISOR_MODULE
 OBJECT MODULE PLACED IN :Fl:superv.OBJ
 COMPILER INVOKED BY: plm86 :Fl:superv.p86

```

$title(' SUPERVISOR TASK FOR AP NOTE 110, OCTOBER 1980')
$large debug
SUPERVISOR_MODULE:
do;
$include(:fl:nuclus.ext)
=      SSAVE NOLIST

089  1      declare token          literally 'word';

          /*****
          /* The six mailboxes immediately following will */
          /* form all of the inter-task communications    */
          /* interfaces for the five tasks that run in     */
          /* this system.                                  */
          *****/

090  1      declare th_in_mbx        token;
091  1      declare th_out_mbx       token;
092  1      declare to_input_mbx     token public;
093  1      declare to_fft_mbx       token public;
094  1      declare to_output_mbx    token public;
095  1      declare to_supervisor_mbx token public;

096  1      declare return_th_in_mbx token;
097  1      declare return_th_out_mbx token;
098  1      declare dummy_mbx        token;
099  1      declare frame_segment_one token;
100  1      declare frame_segment_two token;
101  1      declare frame_segment_three token;
102  1      declare th_in_segment_token token;
103  1      declare th_out_segment_token token;

104  1      declare general_index      byte;
105  1      declare number_of_fft_data_segments byte;

106  1      declare insert_text_pointer pointer;

107  1      declare abort_flag         word;
108  1      declare status            word;
109  1      declare segment_deleted_tally word;
110  1      declare text_length       word;

111  1      declare root_job_token     token;
112  1      declare input_task_token  token;
113  1      declare fft_task_token     token;
114  1      declare output_task_token token;

115  1      declare parameters        structure(
          actual_samples              word,

```

```

actual_interval word,
frequency_answer(5) byte,
actual_frames_to_average word,
frames_to_average_answer(5) byte,
continuous_flag word,
continuous_flag_answer(5) byte);

116 1 declare abort literally 'OFFH';
117 1 declare ascii_9 literally '039H';
118 1 declare carriage_return literally '0DH';
119 1 declare forever literally 'while 1';
120 1 declare line_feed literally '0AH';
121 1 declare new_fft_run literally 'OFFH';
122 1 declare no_abort literally '00H';
123 1 declare no_response_requested literally '00H';
124 1 declare null literally '00H';
125 1 declare queue_fifo literally '00H';
126 1 declare queue_priority literally '01H';
127 1 declare rootjob literally '03H';
128 1 declare run_continuous literally 'OFFFFH';
129 1 declare size_120_bytes literally '120';
130 1 declare space literally '20H';
131 1 declare supervisor_job literally '00H';
132 1 declare th_read literally '01H';
133 1 declare th_write literally '05H';
134 1 declare wait_forever literally 'OFFFFH';

/* The following declaration sets the characters */
/* to send the cursor home at the beginning of */
/* each message to the display screen. The */
/* sequence is tilde, DC2 (Hazeltime)). */

135 1 declare cursor_home_chars(2) byte data (07EH,012H);

136 1 declare frame_pointer pointer;
137 1 declare frame_pointer_values structure(
    offset word,
    base word) at (@frame_pointer);

138 1 declare frame_based_frame_pointer structure(
    samples_per_frame word,
    sample_interval word,
    frames_to_average word,
    continuous_flag word,
    this_frame_number word,
    number_samples_missed word,
    sample_pointer word,
    reset_flag word,
    sample(256) integer);

139 1 declare th_in_segment_pointer pointer;
140 1 declare th_in_segment_pointer_values structure(
    offset word,
    base word) at (@th_in_segment_pointer);

```

```

141 1 declare th_in_segment_based_th_in_segment_pointer
      structure(
        function word,
        count word,
        exception_code word,
        actual_continuous word,
        message(112) byte);

142 declare th_out_segment_pointer pointer;
143 declare th_out_segment_pointer_values structure(
      offset word,
      base word) at (@th_out_segment_pointer);

144 declare th_out_segment_based_th_out_segment_pointer
      structure(
        function word,
        count word,
        exception_code word,
        actual word,
        home_chars(2) byte,
        line_index(24) byte,
        message(84) byte);

145 declare frequency_question(*) byte data ('Please'
      ' enter the highest frequency in Hz (120,
      ' 600, 1200, 6000, OR 12000):');

146 declare frequency_answers data(*) byte data (05H,
      03H,03H,04H,04H,05H,0H,
      '120,600,1200,6000,12000',
      42H,0FH,000DH,03H,087H,01H,
      4EH,00H,027H,00H,00H,000H);

147 declare average_question(*) byte data ('Please'
      ' enter the number of frames to average'
      ' (1,2,4,8,16, OR 32):');

148 declare average_answers data(*) byte data (06H,
      01H,01H,01H,01H,02H,02H,
      '1,2,4,8,16,32',
      01H,00H,02H,00H,04H,00H,
      08H,00H,10H,00H,20H,00H);

149 declare continuous_question(*) byte data ('Please'
      ' enter 'I' for one sample run or 'C'
      ' for continuous running:');

150 declare continuous_answers data(*) byte data (03H,
      01H,01H,01H,00H,00H,00H,
      '1 C C',
      00H,00H,0FFH,0FFH,0FFH,0FFH,
      00H,00H,00H,00H,00H,00H);

151 declare reject_message(*) byte data ('I cannot'
      ' accept your answer. PLEASE TRY AGAIN.');
```

```

152 1 declare status_line_one(*) byte data ('Current'
    ' settings are the following: frequency'
    ' range 0 to Hz,');

153 1 declare status_line_two(*) byte data ('
    ' frames to average per output display,'
    ' and ');

154 1 declare continuous_runs(*) byte data
    ('continuous runs.');
```

```

155 1 declare single_run(*) byte data
    ('a single run.');
```

```

156 1 declare go_ahead_question(*) byte data ('If '
    ' these settings are correct, enter "G"'
    ' to begin running.');
```

```

157 1 declare header_line_one(*) byte data (' INTEL'
    ' APNOTE 110--THE iRMX 86 OPERATING SYSTEM'
    ' AND iAPX 86 COMPONENT DESIGNS.');
```

```

/* The following three procedure declarations */
/* link the tasks outside the SUPERVISOR MODULE */
/* with the supervisor task. */
*****
```

```

158 1 INPUT_TASK: PROCEDURE EXTERNAL;
159 2 END INPUT_TASK;

160 1 FFT_TASK: PROCEDURE EXTERNAL;
161 2 END FFT_TASK;

162 1 OUTPUT_TASK: PROCEDURE EXTERNAL;
163 2 END OUTPUT_TASK;

*****
/* The following five procedures are general */
/* utility procedures called by the primary */
/* working procedures. */
*****
```

```

/* Blank line just fills the message buffer */
/* with blank characters. */
*****
```

```

164 1 BLANK_LINE: PROCEDURE;

165 2 declare blank_line_index word;
```



```

166 2 do blank_line_index = 0 to 78;
167 3   th_out_segment.message(blank_line_index)
      = space;
168 3   end;
169 2   th_out_segment.message(79) = carriage_return;
170 2   END BLANK_LINE;

      /*****
      /* DISPLAY LINE sends the message to the */
      /* terminal handler output mailbox and */
      /* waits for the segment to be returned. */
      /*****/

171 1   DISPLAY_LINE: PROCEDURE;
172 2   call rq$send$message (th_out_mbx,
      th_out_segment.token,
173 2   th_out_segment.token = rq$receive$message
      (return th_out_mbx,
      wait_forever, @dummy_mbx,
      @status);
      /*****/
174 2   END DISPLAY_LINE;

      /*****/
      /* INSERT TEXT fills the output message segment */
      /* with the chosen message and pads the rest of */
      /* the line with blanks. */
      /*****/

175 1   INSERT_TEXT: PROCEDURE(TEXT_POINTER, HOW_MANY);
176 2   declare text_pointer pointer;
177 2   declare how_many word;
178 2   declare dummy_based_text_pointer_structure(
      entries(80) byte);
179 2   declare insert_text_index word;
180 2   do insert_text_index = 0 to (how_many - 1);
181 3   th_out_segment.message(insert_text_index) =
      dummy_based_text_pointer_structure(insert_text_index);
182 3   end;
183 2   do while insert_text_index < 79;
184 3   th_out_segment.message(insert_text_index)
      = space;
185 3   insert_text_index = insert_text_index + 1;
186 3   end;
187 2   th_out_segment.message(79) = carriage_return;
188 2   END INSERT_TEXT;

```

```

*****
/* Move_down_line puts the required number of */
/* line_feed_characters in the first 6th */
/* through 29th character of the output */
/* message. Each line is displayed on the */
/* screen in its proper location. This allows */
/* multiple tasks to access the screen */
/* without having to blank the line each */
/* time. This technique assumes each message */
/* sends the cursor home each time. */
*****

189 1 MOVE_DOWN_LINE: PROCEDURE (SKIP_LINES);
190 2 declare skip_lines word;
191 2 declare move_down_line_index word;

192 2 if skip_lines > 0 then
193 2 do move_down_line_index = 0 to (skip_lines - 1);
194 3 th_out segment.line_index (move_down_line_index)
    = line_feed;
195 3 end;

196 2 do move_down_line_index = skip_lines to 23;
197 3 th_out segment.line_index (move_down_line_index)
    = null;
198 3 end;

199 2 END MOVE_DOWN_LINE;

/* ***** */
/* SEND REJECT MESSAGE is called by INPUT */
/* PARAMETERS. It just sends a reject message */
/* to the CRT to inform the user that the */
/* answer the user gave was not valid. */
/* ***** */

200 1 SEND_REJECT_MESSAGE: PROCEDURE;
201 2 call move_down_line (21);
202 2 text_length = size(reject_message);
203 2 insert_text_pointer = @reject_message;
204 2 call insert_text (insert_text_pointer, text_length);
205 2 call display_line;
206 2 END SEND_REJECT_MESSAGE;

/* ***** */
/* The following procedures are the primary */
/* working procedures called by the supervisor */
/* procedure and its called procedures. */
/* ***** */

```

```

*****/******
/* PURGE MAILBOX removes all tokens from */
/* a mailbox. The purpose is to remove segments */
/* waiting for processing by one of the tasks */
/* if the operator has specified an abort */
/* request. If the segment deletion was */
/* successful, PURGE MAILBOX updates the */
/* segment deleted tally. */
/******
207 1 *****PURGE_MAILBOX: PROCEDURE(MAILBOX_TO_PURGE);
208 2 declare mailbox_to_purge token;
209 2 declare for_110_milliseconds literally '0BH';
210 2 declare message_received_b literally '0H';
211 2 declare contents_token token;
212 2 declare purge_dummy_mbx token;
213 2 declare purge_status token;
214 2 purge_status = message_received;
215 2 do while purge_status = message_received;
216 3 contents_token = rq$receive$message
    (mailbox_to_purge, for_110_milliseconds,
    @purge_dummy_mbx, @purge_status);
217 3 if purge_status = message_received then
218 3 do;
219 4 call rq$delete$segment
    (contents_token, @purge_status);
220 4 segment_deleted_tally =
    segment_deleted_tally + 1;
221 4 purge_status = message_received;
222 4 end;
223 3 end;
224 2 END PURGE_MAILBOX;

*****/******
/* MONITOR MAILBOXES polls the */
/* return_th in mailbox and the */
/* to_supervisor_mailbox for messages. The */
/* messages will be abort (from the operator), */
/* and FFT done or OUTPUT done if the runs are */
/* not continuous. If the runs are not */
/* continuous and an FFT done message is */
/* received, MONITOR_MAILBOXES will initialize */
/* the OUTPUT task. */
/******
225 1 *****MONITOR_MAILBOXES: PROCEDURE;

```

```

226 2      declare cannot_wait      literally '00H';
227 2      declare done              literally 'OFFH';
228 2      declare for_400_milliseconds  literally '28H';
229 2      declare message_received  literally '00H';
230 2      declare not_done          literally '00H';

231 2      declare monitor_dummy_mbx token;
232 2      declare monitor_token     token;
233 2      declare monitor_status    token;

234 2      declare done_flag         byte;
235 2      declare monitor_index     word;

236 2      done_flag = not_done;

237 2      segment deleted_tally = 0;
238 2      call rq$send$message
      (th in_mbx, th in_segment_token,
      return_th in_mbx, @status);
239 2      do while done_flag = not done;
      /* Check for operator input here. */
240 3      monitor_token = rq$receive$message
      (return_th in_mbx, for_400_milliseconds,
      @monitor_dummy_mbx, @monitor_status);
241 3      if monitor_status = message_received then
242 3      do while segment_deleted_tally <
      number_of_fft_data_segments;
243 4      call purge_mailbox (to_fft_mbx);
244 4      if segment_deleted_tally <
      number_of_fft_data_segments then
245 4      call purge_mailbox (to_input_mbx);
246 4      if segment_deleted_tally <
      number_of_fft_data_segments then
247 4      call purge_mailbox (to_supervisor_mbx);
248 4      if segment_deleted_tally <
      number_of_fft_data_segments then
249 4      call purge_mailbox (to_output_mbx);
250 4      done_flag = done;
251 4      end;
252 3      if done_flag = not done then
253 3      do;
254 4      monitor_token = rq$receive$message
      (to_supervisor_mbx, for_400_milliseconds,
      @monitor_dummy_mbx, @monitor_status);
255 4      if monitor_status = message_received then
256 4      do;
257 5      if parameters.continuous_flag =
      run_continuous then
258 5      call rq$send$message
      (to_input_mbx, monitor_token,
      no_response_requested,
      @monitor_status);
      else

```

```

259 5      do;
260 6      call rq$delete$segment
      (monitor token, @monitor_status);
261 6      segment_deleted_tally
      = segment_deleted_tally + 1;
262 6      if segment_deleted_tally
      = number_of_fft_data_segments
      then done_flag = done;
264 6      end;
265 5      end;
266 4      end;
267 3      end;
268 2      END MONITOR_MAILBOXES;

      /*****
      /* SET_SEGMENT initializes the common parameter */
      /* areas of the segment. The pointer to the */
      /* proper segment is set up by */
      /* INITIALIZE_SEGMENTS. */
      *****/

269 1      SET_SEGMENT: PROCEDURE;
270 2      frame.samples_per_frame = 128;
271 2      frame.sample_interval
      = parameters.actual_interval;
272 2      frame.frames_to_average
      = parameters.actual_frames_to_average;
273 2      frame.continuous_flag = parameters.continuous_flag;
274 2      frame.this_frame_number = 00H;
275 2      frame.number_samples_missed = 00H;
276 2      frame.sample_pointer = 00H;
277 2      frame.reset_flag = 00H;
278 2      END SET_SEGMENT;

      /*****
      /* INITIALIZE_SEGMENTS creates the three FFT */
      /* data segments and calls SET_SEGMENT for each */
      /* segment to initialize the common parameter */
      /* areas of the segments. */
      *****/

279 1      INITIALIZE_SEGMENTS: PROCEDURE;
280 2      declare size_528_bytes    literally '528';
281 2      frame_pointer_values.offset = 0;
282 2      frame_segment_one = rq$create$segment
      (size_528_bytes, @status);
283 2      frame_pointer_values.base = frame_segment_one;
284 2      call set_segment;

```

```

285 2      frame.reset_flag = new_fft_run;
286 2      number_of_fft_data_segments = 1;
287 2      call rq$send$message
          (to_input_mbx, frame_segment_one,
           no_response_requested, @status);

288 2      if parameters.actual_frames_to_average > 1 then
289 2          do;
290 3          frame_segment_two = rq$create$segment
          (size_528_bytes, @status);
291 3          frame_pointer_values.base = frame_segment_two;
292 3          call set segment;
293 3          number_of_fft_data_segments = 2;
294 3          call rq$send$message
          (to_input_mbx, frame_segment_two,
           no_response_requested, @status);

295 3          end;

296 2      if parameters.actual_frames_to_average > 2 then
297 2          do;
298 3          frame_segment_three = rq$create$segment
          (size_528_bytes, @status);
299 3          frame_pointer_values.base
          = frame_segment_three;
300 3          call set segment;
301 3          number_of_fft_data_segments = 3;
302 3          call rq$send$message
          (to_input_mbx, frame_segment_three,
           no_response_requested, @status);

303 3          end;

304 2      END INITIALIZE_SEGMENTS;

          /******
          /* INPUT_PARAMETERS contains three procedures: */
          /* set_question_pointers, get_answer, */
          /* and_verify_answers. The INPUT_PARAMETERS */
          /* loop consists of calls to these three */
          /* procedures and, as usual, exists at the end */
          /* of the procedure. */
          /******

305 1      INPUT_PARAMETERS: PROCEDURE;
306 2          declare actual_pointer = pointer;
307 2          declare answer_pointer = pointer;
308 2          declare answer_display_pointer = pointer;
309 2          declare question_pointer = pointer;

310 2          declare answer_actual_value based
          actual_pointer word;

311 2          declare answer_overlay based
          answer_pointer structure(
          number_of_answers byte,

```



```

length_of_answer(6) byte,
values_to_match(30) byte,
really_are(6) word);
312 2 declare answer_display based
      answer_display_pointer structure(
      characters(5) byte);
313 2 declare answer_byte_index byte;
314 2 declare answer_index byte;
315 2 declare answer_match byte;
316 2 declare byte_match byte;
317 2 declare input_byte_index byte;
318 2 declare output_byte_index byte;
319 2 declare question_number byte;
320 2 declare stop_byte byte;

321 2 declare ascii_small_g literally '067H';
322 2 declare ascii_capital_G literally '047H';
323 2 declare average_entry_point literally '0';
324 2 declare continuous_entry_point literally '48';
325 2 declare frequency_entry_point literally '58';
326 2 declare match literally '0FFH';
327 2 declare no_match literally '00H';
328 2 declare not_negative literally '< 255';
329 2 declare nothing_returned literally '00H';

330 2 set_question_pointers: procedure;
331 3 do case question_number;

332 4 do;
333 5 text_length = size (frequency_question);
334 5 question_pointer = @frequency_question;
335 5 answer_pointer = @frequency_answers_data;
336 5 actual_pointer = @parameters.actual_interval;
337 5 answer_display_pointer = @parameters.frequency_answer;
338 5 end;

339 4 do;
340 5 text_length = size (average_question);
341 5 question_pointer = @average_question;
342 5 answer_pointer = @average_answers_data;
343 5 actual_pointer = @parameters.actual_frames_to_average;
344 5 answer_display_pointer = @parameters.frames_to_average_answer;
345 5 end;

346 4 do;
347 5 text_length = size (continuous_question);
348 5 question_pointer = @continuous_question;
349 5 answer_pointer = @continuous_answers_data;
350 5 actual_pointer = @parameters.continuous_flag;

```

```

351 5      answer_display_pointer
352 5      end;
353 4      end;
354 3      end set_question_pointers;

355 2      /* get_answer: procedure;
          /* First display the question to be answered */
          /* by the operator. */
356 3      call insert_text(question_pointer, text_length);
357 3      call move_down_line (19);
358 3      call display_line;
          /* Then blank the line below for an answer line. */
359 3      call blank_line;
360 3      call move_down_line (20);
361 3      call display_line;
          /* Now wait for a response from the operator. */

362 3      call rq$send$message
          (th_in_mbx, th_in_segment token,
           return_th_in_mbx, @status);
363 3      th_in_segment_token = rq$receive$message
          (return_th_in_mbx, wait_forever,
           @dummy_mbx, @status);
364 3      th_in_segment_pointer values.base
          = th_in_segment_token;
          /* If there is no message returned then send */
          /* a reject message. */
365 3      if th_in_segment.actual = nothing_returned
          then call send_reject_message;
          /* Otherwise it is time to check the response */
          /* against the possible answers. */
          else
367 3      do;
368 4      answer_match = no_match;
          /* Set the number of possible answers. */
369 4      answer_index
          = answer_overlay.number_of_answers;
          /* Start a loop to check all of the */
          /* possible answers. */

```

```

370 4      do while (answer_match = no_match) and (answer_index > 0);
          /* Set the starting point for the */
          /* byte by byte compare. */
          /* Set question pointers */
371 5      answer_byte_index = (answer_index * 5) - 1;

          /* Set the stopping point for */
          /* the compare. */

          /* First display the question to be answered */
372 5      stop_byte = answer_byte_index
              - answer_overlay.length_of_answer
              (answer_index - 1);
          /* Start with a "match" so we can */
          /* check until "no match" occurs. */
          /* Then blank the line below for an answer line. */
373 5      byte_match = match;

          /* Set starting point at the right end */
          /* of the input data (allows us to */
          /* ignore leading blanks and the */
          /* ending carriage return). */

374 5      input_byte_index = th_in_segment.actual-2;

          /* Scan the bytes until all pertinent */
          /* ones are checked or a "no_match" */
          /* occurs. */

375 5      do while (byte_match = match) and
          (answer_byte_index > stop_byte);
376 6      if (input_byte_index not negative)
          then do;
378 7      if th_in_segment.message
          (input_byte_index) =
          answer_overlay.values_to_match
          (answer_byte_index)
          then byte_match = match;
          else byte_match = no_match;
380 7      /* Set the number of possible answers. */
381 7      /* against the possible answers. */
382 6      else byte_match = no_match;

          else
383 6      answer_byte_index = answer_byte_index - 1;
384 6      input_byte_index = input_byte_index - 1;
385 6      end;

          /* Set the number of possible answers. */
          /* A "match" at this point means ALL */
          /* bytes matched. */

          /* Set the number of possible answers. */
386 5      if byte_match = match then
387 5      do;
          /* Set real values via */
          /*

```

```

/*      blank, so blank */answer_actual*value overlay.      */
/*      the reject message line and reset
388 6      answer_actualvalue
/*      answer overlay.really_are
/*      (answer_index - 1);
389 6      answer_match = match;
/*      question number = 1;
/*      insert displayable values for */
/*      later display.      */
/*      call display_line;
390 6      answer_byte_index = 4;
391 6      input_byte_index = (answer_index*5)-1;

392 6      while answer_byte_index not negative;
393 7      answer_display_characters
/*      (answer_byte_index)
/*      answer overlay.values_to_match
/*      (input_byte_index);
394 7      input_byte_index
/*      = input_byte_index - 1;
395 7      answer_byte_index
/*      = answer_byte_index - 1;
396 7      end;

/* We got a match, so be sure the */
/* reject message line is blanked. */

397 6      call move_down_line(21);
398 6      call blank_line;
399 6      call display_line;
400 6      call insert_text(status_line_one, text_length);

/* If no match, then let's compare the */
/* input with the next possible answer. */

401 5      else answer_index = answer_index - 1;
402 + 5      end;
/* If we got a match, then we can move on */
/* to the next question.      */
403 4      if answer_match = match then
/*      question_number = question_number + 1;
/*      Otherwise we have to check for an */
/*      abort request of '99' */
/*      call move_down_line;
/*      call display_line;

else
405 4      do;
406 5      input_byte_index = th_in_segment.actual-2;
407 5      if (th_in_segment.message(input_byte_index)
/*      text_length = size(status_line_two);
/*      call insert_text(status_line_two, text_length);
/*      and
/*      (th_in_segment.message(input_byte_index -
1) /*      We have to insert the displayable "frames"
/*      = ascii_9) then

```

```

/* Abort requests are valid, so blank */
/* the reject message line and reset */
/* the question number so we start */
/* asking all over. */
408 5      do;
409 6          question_number = 1;
410 6          call move_down_line (21);
411 6          call blank_line;
412 6          call display_line;
413 6      end;
/* But if nothing matched and the */
/* answer was not an abort request, */
/* then we have to ask the operator */
/* to try again on this question. */
414 5      call send_reject_message;
415 5      end;
416 4      end;
417 3      end get_answer;

/* We got a match, so we are the */
418 2 verify_answers: procedure;

/* First put the output line in the buffer. */
419 3      text_length = size (status_line_one);
420 3      call insert_text (@status_line_one, text_length);

/* If no match, then let's compare the */
/* Then insert the displayable frequency answer. */
421 3      input_byte_index = 0;
422 3      stop_byte = frequency_entry_point + 4;
423 3      do output_byte_index = frequency_entry_point to stop_byte;
424 4          th_out_segment.message (output_byte_index) =
              parameters.frequency_answer (input_byte_index);
425 4          input_byte_index = input_byte_index + 1;
426 4      end;
/* Otherwise we have to check for an */
/* abort */
427 3      call move_down_line (19);
428 3      call display_line;

/* We have sent the first line, now it is time */
/* to get the second line. */
429 3      text_length = size (status_line_two);
430 3      call insert_text (@status_line_two, text_length);
/* We have to insert the displayable "frames */

```

```

/* to average" answer. */
431 3    input_byte_index = 0;
432 3    stop_byte = average_entry_point + 4;
433 3    do output_byte_index
        = average_entry_point to stop_byte;
434 4    th_out.segment.message (output_byte_index) =
        parameters.frames_to_average_answer
        (input_byte_index);
435 4    input_byte_index = input_byte_index + 1;
436 4    end;

/* The continuous answer is different--we have */
/* to decide if we have continuous runs or */
/* single runs, and insert those words in the */
/* display line. */

437 3    input_byte_index = 0;
438 3    stop_byte = continuous_entry_point + 15;
439 3    if parameters.continuous_flag = run_continuous
        then
440 3        do output_byte_index
            = continuous_entry_point to stop_byte;
441 4        th_out.segment.message (output_byte_index) =
            continuous_runs (input_byte_index);
442 4        input_byte_index = input_byte_index + 1;
443 4        end;
        else
444 3        do output_byte_index
            = continuous_entry_point to stop_byte;
445 4        th_out.segment.message (output_byte_index) =
            single_run (input_byte_index);
446 4        input_byte_index = input_byte_index + 1;
447 4        end;

/* Then send the message and wait for a response */
/* from the operator. */

448 3    call move_down_line(20);
449 3    call display_line;

450 3    text_length = size (go_ahead_question);
451 3    call insert_text (@go_ahead_question, text_length);
452 3    call move_down_line (21);
453 3    call display_line;

454 3    call blank_line;
455 3    call move_down_line(22);
456 3    call display_line;

457 3    call rq$send$message
        (th_in_mbx, th_in_segment_token,
        return_th_in_mbx, @status);
458 3    th_in_segment_token
        = rq$receive$message

```



```

                                (return_th_in_mbx, wait_forever,
                                @dummy_mbx, @status);
459   3   th_in_segment_pointer values.base
                                = th_in_segment_token;
460   3   input_byte_index = th_in_segment.actual - 2;

/* Check for a "g" or "G" (we aren't fussy). If */
/* we got it, let's quit asking the selection */
/* questions and go. If not, we have to start */
/* at question 1 again rather than try to find */
/* out which of his or her answers wasn't */
/* acceptable. */

461   3   if (th_in_segment.message (input_byte_index)
                                = ascii_small_g) or
                                (th_in_segment.message (input_byte_index)
                                = ascii_capital_G) then;
463   3       else question_number = 0;

464   3   call blank_line;
465   3   call move_down_line (21);
466   3   call display_line;

467   3   end verify_answers;

/* * * * * * */
/* As usual, the actual INPUT PARAMETERS control */
/* loop is at the end. */
/* * * * * * */

468   2   question_number = 0;

/* All we do is get the next question, ask the */
/* question until it is answered successfully, */
/* ask all of the questions, then check all of */
/* the answers. If the operator doesn't like */
/* the set of answers, we loop through them */
/* again. First we make sure the reject message */
/* line and other pertinent lines start out */
/* blanked. */

469   2   call blank_line;
470   2   call move_down_line (18);
471   2   call display_line;
472   2   call move_down_line (21);
473   2   call display_line;
474   2   call move_down_line (22);
475   2   call display_line;
476   2   call move_down_line (23);
477   2   call display_line;

478   2   input_loop: do while question_number < 3;
479   3       call set_question_pointers;
480   3       call get_answer;

```



```

501 2      END INITIALIZE_TASKS;

/* ***** */
/* Initial screen displays the initial two
/* lines on the screen and sends blank lines
/* for all the other lines of the first screen.
/* ***** */

502 2      INITIAL_SCREEN: PROCEDURE;
503 2      declare initial_screen_index word;
504 2      call move_down_line(0);
505 2      call blank_line;
506 2      call display_line;

507 2      call move_down_line(1);
508 2      text_length = size(header_line_one);
509 2      insert_text_pointer = @header_line_one;
510 2      call insert_text(insert_text_pointer, text_length);
511 2      call display_line;
512 2      call blank_line;
513 2      initial_screen_index = 2 to 23;
514 3      call move_down_line(initial_screen_index);
515 3      call display_line;
516 3      end;

517 2      END INITIAL_SCREEN;

/* ***** */
/* INITIALIZE_BUFFERS takes care of the
/* initialization required for general
/* SUPERVISOR TASK start up.
/* ***** */

518 1      INITIALIZE_BUFFERS: PROCEDURE;
519 2      return_th_in_mbx = rq$create$mailbox
520 2      call rq$catalog$object
521 2      return_th_out_mbx = rq$create$mailbox
522 2      call rq$catalog$object
523 2      to_input_mbx = rq$create$mailbox
524 2      call rq$catalog$object
525 2      to_fft_mbx = rq$create$mailbox

```

```

526 2  call rq$catalog$object      (supervisor_job, to_fft_mbx,
    (10, 'TO FFT MBX'), @status);
527 2  to_output_mbx = rq$create$mailbox
    (queue_priority, @status);
528 2  call rq$catalog$object      (supervisor_job, to_output_mbx,
    (10, 'TO OUT MBX'), @status);
529 2  to_supervisor_mbx = rq$create$mailbox
    (queue_priority, @status);
530 2  call rq$catalog$object      (supervisor_job, to_supervisor_mbx,
    (10, 'TO SUP MBX'), @status);

531 2  root_job_token = rq$get$task$tokens
    (root_job, @status);
532 2  th_out_mbx = rq$lookup$object
    (root_job_token, @ (11, 'RQTHNORMOUT'),
    wait_forever, @status);
533 2  th_in_mbx = rq$lookup$object
    (root_job_token, @ (10, 'RQTHNORMIN'),
    wait_forever, @status);

534 2  th_in_segment_token = rq$create$segment
    (size 120 bytes, @status);
535 2  call rq$catalog$object      (supervisor_job, th_in_segment_token,
    (10, 'S THIN SEG'), @status);
536 2  th_in_segment_pointer_values.offset = 0;
537 2  th_in_segment_pointer_values.base
    = th_in_segment_token;
538 2  th_in_segment.function = th_read;
539 2  th_in_segment.count = 82;

540 2  th_out_segment_token = rq$create$segment
    (size 120 bytes, @status);
541 2  call rq$catalog$object      (supervisor_job, th_out_segment_token,
    (11, 'S THOUT SEG'), @status);
542 2  th_out_segment_pointer_values.offset = 0;
543 2  th_out_segment_pointer_values.base
    = th_out_segment_token;
544 2  th_out_segment.function = th_write;
545 2  th_out_segment.count = 111;

546 2  do general_index = 0 to 2;
547 3  th_out_segment.home_chars (general_index)
    = cursor_home_chars (general_index);
548 3  end;
549 2  END INITIALIZE_BUFFERS;

```

```

/****
/* At last, the SUPERVISOR TASK! All it does is */

```

```

/* call other procedures to initialize the */
/* screen, input the parameters, clean up the */
/* old FFT segments from the mailboxes, set up */
/* new segments, create the tasks, and then wait */
/* for messages from the operator (abort) or */
/* other tasks (FFT or OUTPUT done). */
/***** */
550 1 SUPERVISOR_TASK: PROCEDURE PUBLIC;
551 2 call initialize_buffers;
552 2 call rq$send$init$task;

553 2 call initial_screen;
554 2 call initialize_tasks;

555 2 do forever;
556 3 call input_parameters;
557 3 call initialize_segments;
558 3 call monitor_mailboxes;

559 3 end;

560 2 END SUPERVISOR_TASK;
561 1 END SUPERVISOR_MODULE;

MODULE INFORMATION:
CODE AREA SIZE = 1032H 4146D
CONSTANT AREA SIZE = 0000H 0D
VARIABLE AREA SIZE = 0084H 132D
MAXIMUM STACK SIZE = 0024H 63D
1197 LINES READ
0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION

/* At last, the SUPERVISOR TASK! All it does is */

```

```

ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE INPUT_TASK_MODULE
OBJECT MODULE PLACED IN :F1:input.OBJ
COMPILER INVOKED BY: plm86 :F1:input.p86
$title('INPUT_TASK FOR AP NOTE 110, OCTOBER 1980')
$large debug
INPUT_TASK_MODULE:
do;
$include(:f1:nucprm.ext)
= $SAVE NOLIST
89 1 declare token literally 'word';
/* The following two tokens, the FFT sample
/* segment format, and the root job directory
/* form the entire interface for this task with
/* the rest of the system.
90 1 declare to_input_mbx token external;
91 1 declare to_fft_mbx token external;

92 1 declare ascii_mask literally '30H';
93 1 declare carriage_return literally '0DH';
94 1 declare done literally 'OFFH';
95 1 declare first_loop literally 'OFFH';
96 1 declare forever literally 'while 1';
97 1 declare frames_to_process_entry literally '50';
98 1 declare hardware_interrupt_level_3 literally '0038H';

99 1 declare interrupt_task_created literally 'OFFH';
100 1 declare interrupt_task_not_created literally '00H';
101 1 declare latch_the_data literally '040H';
102 1 declare line_feed literally '0AH';
103 1 declare new_fft_run literally 'OFFH';
104 1 declare no_response_requested literally '00H';
105 1 declare no_data_segment literally '00H';
106 1 declare not_done literally '00H';
107 1 declare not_first_loop literally '00H';
108 1 declare not_valid literally '00H';
109 1 declare null literally '00H';
110 1 declare processed_so_far_entry literally '33';
111 1 declare queue_fifo_value literally '00H';
112 1 declare rootjob literally '03H';
113 1 declare run_continuous literally 'OFFFFH';
114 1 declare sample_LSB literally '0081H';
115 1 declare sample_MSB literally '0080H';
116 1 declare size_2_bytes literally '2';
117 1 declare size_120_bytes literally '120';
118 1 declare supervisor_job literally '00H';
119 1 declare th_write literally '05';
120 1 declare this_is_the_interrupt_task literally '01H';
121 1 declare timer_one_port literally '00D2H';
122 1 declare timer_mode_control_port literally '00D6H';
123 1 declare valid literally 'OFFH';

```



```

124 1 declare wait_forever (literally 'OFFFH';
125 1 declare data_segment_token token;
126 1 declare dummy_mbx token;
127 1 declare from_interrupt_task_mbx token;
128 1 declare handler_dummy_mbx token;
129 1 declare handler_status token;
130 1 declare interrupt_status token;
131 1 declare interrupt_task_token token;
132 1 declare interrupt_message token;
133 1 declare output_buffer_token token;
134 1 declare return_mbx token;
135 1 declare root_job token;
136 1 declare signal_interrupt token;
137 1 declare status token;
138 1 declare to_interrupt_task_mbx token;
139 1 declare th_out_mbx token;
140 1 declare sample_data integer;
141 1 declare sample_input_data structure(
142 1     LSB byte,
143 1     MSB byte) at (@sample_data);
144 1 declare current_timer_value word;
145 1 declare timer_values structure(
146 1     LSB byte,
147 1     MSB byte) at (@current_timer_value);
148 1 declare done_flag byte;
149 1 declare first_input_loop_flag byte;
150 1 declare frames_received byte;
151 1 declare general_index byte;
152 1 declare interrupt_task_flag byte;
153 1 declare sample_valid byte;
154 1 declare timer_threshold word;
155 1 declare value_to_convert word;
156 1 declare converted_value structure(
157 1     first_digit byte,
158 1     second_digit byte);
159 1 /* The following declare is for the home */
160 1 /* characters for the Hazeltine terminals. */
161 1 /* The sequence is tilde, DC2. */
162 1 declare cursor_home_chars(2) byte data (07EH,012H);
163 1 declare output_buffer_pointer pointer;

```

```

155 1 declare output_buffer_pointer values structure(
/* of the 8253 have been asked,
/* SLOW INPUT HANDLER is an interrupt procedure,
/* TASK procedure know
/* the buffer is full.
156 1 declare output_buffer based output_buffer_pointer
structure(
function: word,
count word,
exception_code word,
actual word,
home_chars(2) byte,
line_index(24) byte,
character(80) byte);
157 1 declare data_segment_pointer pointer public;
158 1 declare data_segment_pointer values structure(
offset word,
base word) at
(@data_segment_pointer);

/* The following is the FFT data segment format. */
159 1 declare data_segment based data_segment_pointer
structure(
samples_per_frame word,
sample_interval word,
frames_to_average word,
continuous_flag word,
this_frame_number word,
number_samples_missed word,
sample_pointer word,
reset_flag word,
sample(256) integer);

160 1 declare input_status_line(80) byte data
; The INPUT TASK has processed ',
frames out of frames to'
average. ');
161 1 FAST_INPUT_HANDLER:
PROCEDURE (FFT_SEGMENT_POINTER) EXTERNAL;
162 2 DECLARE FFT_SEGMENT_POINTER POINTER;
163 2 END FAST_INPUT_HANDLER;

/*****
/* SLOW INPUT HANDLER is an interrupt procedure */
/* that receives an interrupt when the 8253 */
/* interval timer counts to zero. The 8253 is */
/* free running, so it starts counting from the */
/* top again. The 8253 counter is tested */
/* in a polling fashion to be sure it reads the */
/* sample, which resets the conversion, */
/* at a precise time. This aids in removing
*****/

```

```

/* jitter from the sample intervals. When all */
/* of the samples have been taken, */
/* SLOW_INPUT_HANDLER calls signal interrupt, */
/* which lets the INTERRUPT_TASK procedure know */
/* the buffer is full. */
/* ***** */

164 1 SLOW_INPUT_HANDLER: PROCEDURE INTERRUPT 59 PUBLIC;

/* First set the sample to not valid (we have */
/* to be past the timer threshold before the */
/* sample becomes valid). */

165 2 sample_valid = not_valid;
166 2 timer_loop:

/* Make the timer value stable and read it. */
output (timer_mode_control_port) = latch_the_data;
167 2 timer_values.LSB = input (timer_one_port);
168 2 timer_values.MSB = input (timer_one_port);

/* If it is not past the threshold, then some */
/* future sample will be valid. */

169 2 if current_timer_value > timer_threshold then
170 2 do;
171 3 sample_valid = valid;
172 3 goto timer_loop;
173 3 end;

/* We get to the else only if we are past the */
/* timer threshold. */

else
174 2 do;
175 3 sample_input_data.LSB = input(sample_LSB);
176 3 sample_input_data.MSB = input(sample_MSB);

/* If the sample is valid, we must have come */
/* in before the threshold so we know we */
/* sampled as close to the right time as */
/* possible. */

177 3 if sample_valid = valid then
178 3 do;

/* ***** */
/* However, we want to ignore the first */
/* sample (which was started a long */
/* time ago). */

179 4 if first_input_loop_flag = first_loop then
180 4 first_input_loop_flag = not_first_loop;
/* else */
181 4 do;

```

```

182 5      data_segment.sample
          (data_segment.sample_pointer)
183 5      = sample_data;
          data_segment.sample_pointer
184 5      = data_segment.sample_pointer+1;
185 4      end;
          else
186 3      do;
187 4      data_segment.number_samples_missed
          = data_segment.number_samples_missed+1;
188 4      data_segment.sample_pointer = 0;
189 4      end;
190 3      sample_valid = not valid;
191 3      end;

          /* If we are done, we have to let the */
          /* INPUT TASK know the buffer is full. */
          /* Otherwise, we wait for the next interrupt. */

192 2      if data_segment.sample_pointer
          >= data_segment.samples_per_frame then
193 2      call rq$signal$interrupt
          (hardware_interrupt_level_3,
           @handler_status);

          else
194 2      call rq$exit$interrupt
          (hardware_interrupt_level_3,
           @handler_status);

195 2      END SLOW_INPUT_HANDLER;

          /******
          /* INTERRUPT TASK exists because if an interrupt */
          /* task goes to sleep, the level of the */
          /* interrupt task, interrupt handler, and lower */
          /* levels remain disabled. In order to prevent */
          /* this from happening in this application, this */
          /* task notifies the INPUT TASK that the buffer */
          /* is full. INPUT TASK disables Level 3 and */
          /* returns the token to INTERRUPT TASK. */
          /* INTERRUPT TASK will then call wait interrupt, */
          /* enabling lower levels. Since INPUT TASK */
          /* disabled Level 3, no Level 3 interrupts will */
          /* be serviced until Level 3 is enabled by */
          /* INPUT TASK. */
          /* NOTE THAT PLM/86 REQUIRES THE USE OF THE */
          /* BUILT IN INTERRUPT$PTR PROCEDURE TO OBTAIN */
          /* THE PROPER INTERRUPT PROCEDURE ENTRY POINT. */
          /******

196 1      INTERRUPT_TASK: PROCEDURE PUBLIC;

```

```

197 2    call rq$set$interrupt
        (hardware_interrupt_level_3,
        this_is_the_interrupt_task,
        INTERRUPT$PTR(SLOW_INPUT_HANDLER),
        no_data_segment, @interrupt_status);
198 2    do forever;
199 3    call rq$wait$interrupt
        (hardware_interrupt_level_3,
        @interrupt_status);
200 3    call rq$send$message
        (from_interrupt_task_mbx,
        interrupt_message_token,
        to_interrupt_task_mbx, @interrupt_status);
201 3    interrupt_message_token = rq$receive$message
        (to_interrupt_task_mbx, wait_forever,
        @dummy_mbx, @interrupt_status);
202 3    end;
203 2    END INTERRUPT_TASK;

    /******
    /* CONVERT DIGITS is a small procedure for
    /* converting a hex number into an ASCII
    /* number, with the advance knowledge that the
    /* hex number will be less than 99 decimal (in
    /* this case, less than 32 decimal).
    /******

204 1    CONVERT_DIGITS: PROCEDURE;
205 2    converted_value.first_digit = ascii_mask;
206 2    converted_value.second_digit = ascii_mask;
207 2    done_flag = not done;
208 2    do while done_flag = not done;
209 3    value_to_convert = value_to_convert - 10;

    /* The problem here is we need to check for
    /* a negative value when we have BYTE values
    /* which are, by definition, positive and
    /* modulo 256. So we adapt by checking for
    /* > 200 decimal, which should mean the
    /* value has "wrapped" around zero. If it
    /* has, we can get our previous value back
    /* by adding 10.
    /******

210 3    if value_to_convert < 200 then
211 3    converted_value.first_digit

```

```

    else
    do;
    value_to_convert = value_to_convert + 10;
    converted_value.second_digit
    = converted_value.second_digit +
    value_to_convert;
    done_flag = done;
    end;
end;

END CONVERT_DIGITS;

/*****
/* SEND_STATUS converts the current frame
/* number into ASCII and stuffs it into the
/* previously initialized status line. Then
/* SEND_STATUS sends the status line to the
/* terminal handler and waits for the segment
/* to be returned.
*****/

219 1 SEND_STATUS: PROCEDURE;
220 2 value_to_convert = data_segment.this_frame_number;

221 2 call convert_digits;
222 2 output_buffer.character (processed_so_far_entry)
    = converted_value.first_digit;
223 2 output_buffer.character (processed_so_far_entry+1)
    = converted_value.second_digit;
224 2 value_to_convert = data_segment.frames_to_average;
225 2 call convert_digits;
226 2 output_buffer.character (frames_to_process_entry)
    = converted_value.first_digit;
227 2 output_buffer.character (frames_to_process_entry+1)
    = converted_value.second_digit;
228 2 call rq$send$message
    (th_out_mbx, output_buffer_token,
    return_mbx, @status);
229 2 output_buffer_token = rq$receive$message
    (return_mbx, wait_forever,
    @dummy_mbx, @status);
230 2 END SEND_STATUS;

/*****
/* INPUT_DATA selects the fast or slow input
/* handler, initializes the 8253 timer as
/* necessary, and calls the appropriate input
*****/

```



```

/* handler. Please note the values of the */
/* intervals selected for sampling are */
/* scaled by 60/64 so the actual frequency */
/* output of the 128 sample FFT algorithm will */
/* match up with the base 10 x axis labels. */
/* Base 10 doesn't map too well to a binary */
/* x-axis that runs from 1 to 64. */
/***** */
231 1 INPUT_DATA: PROCEDURE;
232 2 declare LSB_120Hz_interval literally '058H';
233 2 declare MSB_120Hz_interval literally '002H';
234 2 declare LSB_600Hz_interval literally '078H';
235 2 declare MSB_600Hz_interval literally '000H';
/* The 8253 timer is running at 143.6 Khz, or */
/* 6.5 microseconds per count. We have to */
/* restart the sampling process precisely, so */
/* we count down after the interrupt to be */
/* sure we are synchronized. In this case, */
/* we have a 300 microsecond window after the */
/* interrupt to get the sample. 300 */
/* microseconds is roughly 42 times 6.5 */
/* microseconds. */
236 2 declare threshold_for_120Hz literally '022EH';
237 2 declare threshold_for_600Hz literally '0048H';
238 2 declare a_3906_microsecond_interval
literally '0F42H';
239 2 declare five_places literally '5';
240 2 declare nucleus_allocated_stack literally '00H';
241 2 declare shift_integer_right literally 'SAL';
242 2 declare software_priority_level_0 literally '00H';
243 2 declare software_priority_level_66 literally '66';
244 2 declare stack_size_512 literally '512';
245 2 declare task_flags literally '00H';
246 2 declare this_task literally '00H';
247 2 declare timer_mode_control_word literally '74H';
248 2 declare enable_conversion literally '00';
249 2 declare input_command literally '0080H';
/* The first thing we do is start the conver- */
/* sions. We don't care about the first */
/* data since we are going to ignore it. Each */
/* time we read both bytes of the present */
/* converted value, we start the next */
/* conversion. This initialization will */
/* prepare for the real data gathering. */
250 2 output(input_command) = enable_conversion;

```

```

251 2      if data_segment.sample_interval > 391 then
252 2          do;
253 3          output (timer_mode_control_port)
          /* = timer_mode_control_word;
254 3          /* if data_segment.sample_interval
          /* = a_3906_microsecond_interval then
255 3          do;
256 4          timer_threshold
          /* = threshold_for_120Hz;
257 4          output (timer_one_port)
          /* = LSB_120Hz_interval;
258 4          output (timer_one_port)
          /* = MSB_120Hz_interval;
259 4          end;
          else
260 3          do;
261 4          timer_threshold
          /* = threshold_for_600Hz;
262 4          output (timer_one_port)
          /* = LSB_600Hz_interval;
263 4          output (timer_one_port)
          /* = MSB_600Hz_interval;
264 4          end;
265 3          first_input_loop_flag = first_loop;
266 3          if interrupt_task_flag
          /* = interrupt_task_not_created then
267 3          do;
268 4          interrupt_task_token = rq$create$task
          /* (software_priority_level_66,
          /* @interrupt_task, no_data_segment,
          /* nucleus_allocated_stack,
          /* stack_size_512, task_flags, @status);
269 4          call rq$catalog$object
          /* = rq$catalog$object
          /* (supervisor_job, interrupt_task_token,
          /* @ (12, 'INTERRUPTTSK'), @status);
270 4          interrupt_task_flag
          /* = interrupt_task_created;
271 4          end;
272 3          else call rq$enable
          /* (hardware_interrupt_level_3, @status);
          /* *****
          /* Now we wait until the slow handler */
          /* *****
          /* fills the buffer.
          /* *****
          /* UPDATE FRAME
          /* *****
          /* number parameter on the data segments.
          /* *****
273 3          signal interrupt_token = rq$receive$message
          /* (from interrupt task mbx, wait_forever,
          /* @dummy_mbx, @status);
          /* *****
          /* If we get the token, we know the buffer
          /* is full, so we disable level 3
          /* *****

```

```

274 3      call rq$disable (hardware_interrupt_level_3, @status);
      /* And return the token so the
      /* INTERRUPT TASK can enable lower
      /* interrupt levels. */

275 3      call rq$send$message (to_interrupt_task_mbx,
      signal_interrupt_token,
      no_response_requested, @status);

276 3      end;
277 2      else
      do;

      /* The fast INPUT handler must sample at
      /* precise intervals that do not allow
      /* variable interrupt latency. Therefore
      /* we raise the priority level to 0--the
      /* highest--and just sample in a polling
      /* fashion until the buffer is filled. */

278 3      call rq$set$priority (this task, software_priority_level_0,
      @status);
279 3      call FAST_INPUT_HANDLER (data_segment_pointer);
280 3      call rq$set$priority (this task, software_priority_level_66,
      @status);
281 3      end;
282 2      do general_index = 0 to 127;
283 3          data_segment.sample (general_index)
      = shift integer right
      (data_segment.sample (general_index),
      five_places);
284 3      end;
285 2      do general_index = 128 to 255;
286 3          data_segment.sample (general_index) = 0000H;
287 3      end;
288 2      END INPUT_DATA;

      /* *****
      /* UPDATE_FRAME_NUMBER just updates the frame */
      /* number parameter on the data segments. */
      /* *****
289 1      UPDATE_FRAME_NUMBER: PROCEDURE;
290 2      data_segment.number_samples_missed = 0;
291 2      data_segment.sample_pointer = 0;

```

```

292 2      if frames_received = data_segment.frames_to_average
           then frames_received = 0;
294 2      if data_segment.reset_flag = new_fft_run then
295 2          do;
296 3              frames_received = 0;
297 3              data_segment.reset_flag = 0;
298 3          end;

299 2      frames_received = frames_received + 1;
300 2      data_segment.this_frame_number = frames_received;
301 2      END UPDATE_FRAME_NUMBER;

           /*****
           /* INITIALIZE_BUFFERS takes care of the usual
           /* trivia of setting up the pointers, creating
           /* the return mailbox, looking up the terminal
           /* handler, and all that other small garbage.
           *****/

302 1      INITIALIZE_BUFFERS: PROCEDURE;
303 2          return_mbx = rq$create$mailbox
           (queue fifo, @status);
304 2          call rq$catalog$object
           (supervisor_job, return_mbx,
           @('9','I RET MBX'), @status);

305 2          from_interrupt_task_mbx = rq$create$mailbox
           (queue fifo, @status);
306 2          call rq$catalog$object
           (supervisor_job, from_interrupt_task_mbx,
           @('12','FM INTSK MBX'), @status);

307 2          to_interrupt_task_mbx = rq$create$mailbox
           (queue fifo, @status);
308 2          call rq$catalog$object
           (supervisor_job, to_interrupt_task_mbx,
           @('12','TO INTSK MBX'), @status);

309 2          interrupt_message_token = rq$create$segment
           (size 2 bytes, @status);
310 2          call rq$catalog$object
           (supervisor_job, interrupt_message_token,
           @('10','INTTSK MSG'), @status);

311 2          interrupt_task_flag
           = interrupt_task_not_created;

312 2          output_buffer_token = rq$create$segment
           (size 120 bytes, @status);
313 2          call rq$catalog$object
           (supervisor_job, output_buffer_token,
           @('10','I BUFF SEG'), @status);

```

```

314 2 output_buffer_pointer_values.offset = 0;
315 2 output_buffer_pointer_values.base
    = output_buffer_token;
316 2 output_buffer.function = th_write;
317 2 output_buffer.count = 110;
318 2 do general_index = 0 to 6;
319 3 output_buffer.home_chars (general_index)
    = cursor_home_chars (general_index);
320 3 end;
321 2 do general_index = 0 to 21;
322 3 output_buffer.line_index (general_index)
    = line_feed;
323 3 end;
324 2 do general_index = 22 to 23;
325 3 output_buffer.line_index (general_index)
    = null;
326 3 end;
327 2 do general_index = 0 to 78;
328 3 output_buffer.character (general_index)
    = input_status_line (general_index);
329 3 end;
330 2 root_job_token = rq$get$task$tokens
    (rootjob, @status);
331 2 th_out_mbx = rq$lookup$object
    (root_job_token, @('RQTHNORMOUT'),
    wait_forever, @status);
332 2 frames_received = 0;
333 2 END INITIALIZE_BUFFERS;

/*****
/* The actual INPUT TASK begins here. It */
/* initializes the buffers to begin things, */
/* then waits forever for the FFT sample */
/* segment. It then samples the data, fills */
/* the FFT data segment, and sends it to the */
/* FFT TASK. The INPUT TASK then updates its */
/* status line, sends it to the terminal */
/* handler, and returns to the mailbox to */
/* wait forever. */
*****/

334 1 INPUT_TASK: PROCEDURE PUBLIC;
335 2 call initialize_buffers;
336 2 data_segment_pointer_values.offset = 0;
337 2 do forever;

```

```

/* Wait forever for an FFT data segment at */
/* the to_input_mbx. */
338 3      data_segment_token = rq$receive$message
          (to_input_mbx, wait_forever,
          @dummy_mbx, @status);
339 3      data_segment_pointer_values.base
          = data_segment_token;

340 3      call update_frame_number;
341 3      call input_data;

342 3      call send_status;

343 3      call rq$send$message
          (to_fft_mbx, data_segment_token,
          no_response_requested, @status);

344 3      end;
345 2      END INPUT_TASK;
346 1      END INPUT_TASK_MODULE;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 070BH 1803D
CONSTANT AREA SIZE = 0000H 0D
VARIABLE AREA SIZE = 0036H 54D
MAXIMUM STACK SIZE = 002AH 42D
754 LINES READ
0 PROGRAM ERROR(S)
END OF PL/M-86 COMPILATION

```

```

16BX 86 register usage is the following:
AX - general
CX - loop delay counter
DX - sample value
BX - address/index
SP - stack
SI - offset index into FFT 81 - not used
DI - data segment
PS - base for FFT data
ES - not used

```


MCS-86 MACRO ASSEMBLER FSTINP
 ISIS-II MCS-86 MACRO ASSEMBLER V2.1 ASSEMBLY OF MODULE FSTINP
 OBJECT MODULE PLACED IN :F1:FSTINP.OBJ
 ASSEMBLER INVOKED BY: asm86 :f1:fstinp.a86

```

LOC  OBJ  LINE  SOURCE
1      ; *****
2      ;
3      ; FAST INPUT HANDLER for APNOTE 110,
      ; OCTOBER 1980
4      ;
5      ; FAST INPUT HANDLER is an assembler routine
      ; that runs at priority
6      ; level 0 and simply drives an analog to
      ; digital convertor and
7      ; stuffs the samples into a data segment until
      ; all of the samples
8      ; have been taken. FAST INPUT HANDLER has
      ; passed to it the address
9      ; of the data segment, in which the offset is
      ; known to be zero.
10     ; FAST INPUT HANDLER returns nothing to the
      ; calling routine.
11     ;
12     ; FAST INPUT HANDLER provides the proper timing
      ; for sampling at a
13     ; 39, 78, and 391 microsecond intervals using
      ; timed loops of
14     ; software instructions. In order to provide
      ; an FFT without large
15     ; amounts of jitter, the sample intervals must
      ; be uniform in time.
16     ; iRMX 86 cannot guarantee this uniformity due
      ; to its real time
17     ; design, so this routine takes complete
      ; control of the processor
18     ; for the (39 times 128) 4.9 milliseconds or
      ; (78 times 128) 9.9
19     ; or (391 times 128) 50 milliseconds required
      ; to complete a frame
20     ; of 128 samples.
21     ;
22     ; iAPX 86 register useage is the following:
23     ;
24     ; AX - general                BX - stack index
25     ; CX - loop delay counter    DX - sample value
26     ;
27     ; BP - stack                  SP - stack
28     ; DI - offset index into FFT SI - not used
29     ; data segment
30     ;
31     ; DS - base for FFT data      ES - not used
32     ; segment

```

```

33 ;*****
34 ;*****
35 ;*****
36 ;*****
37 ;*****
38 ;*****
39 ;*****
40 ;*****
0080 41 SAMPLE_LSB EQU 0080H
0081 42 SAMPLE_MSB EQU 0081H
000E 43 FIRST_PASS EQU 14
0002 44 SAMPLE_INCREMENT EQU 2
0002 45 SAMPLE_INTERVAL EQU 2
010E 46 SAMPLE_MAX EQU 270
47 ;
48 ;
---- 49 FAST_INPUT_DATA SEGMENT WORD
PUBLIC 'DATA'
50 ;
0000 51 LOOP_VALUE DW ?
52 ;
---- 53 FAST_INPUT_DATA ENDS
54 ;
55 ;
---- 56 STACK SEGMENT STACK 'STACK'
57 ;
0000 58 DW 20 DUP(0)
0000 59 ;
---- 60 STACK ENDS
61 ;
62 ;
---- 63 FAST_INPUT_CODE SEGMENT PARA
PUBLIC 'CODE'
64 ;
0000 65 FAST_INPUT_HANDLER PROC FAR
66 ;
0000 1E 67 PUSH DS
0001 55 68 PUSH BP
; SAVE BP IN STACK
0002 8BEC 69 MOV BP, SP
; SET BP TO STACK POINTER
0004 8E5E0A 70 MOV DS, [BP + 10]
; PUT BASE OF SAMPLE SEGMENT IN DS
0007 BF0200 71 MOV DI, SAMPLE_INTERVAL
; DX IS USED TO INDEX INTO THE DATA SEGMENT
000A 8B05 72 MOV AX, DS:[DI]
; SET AX TO SAMPLE INTERVAL PARAMETER
000C BF0E00 73 MOV DI, FIRST_PASS
; RESET DI TO FIRST SAMPLE - 14
000F 3D2700 74 CMP AX, 39
; IF AX = 39, SET LOOP_VALUE TO 9--LOOP
0012 740E 75 JZ SET_39_US

```

```

; TAKES ABOUT 3 US PER DECREMENT
0014 3D4E00 76 CMP AX, 78
; IF AX = 78, SET LOOP_VALUE TO 22--BASIC
0017 7412 77 JZ SET_78_US
; CYCLE_IS_13 US, PLUS (22X3) = 79 US
0019 C70600007E00 R 78 MOV LOOP_VALUE, 126
; 391 IS ONLY ONE LEFT-13 + (126X3)
; = 391
001F EB1090 79 JMP INPUT_LOOP
0022 C70600000900 R 80 SET 39 US: MOV LOOP_VALUE, 9
; TIMING IS BY SOFTWARE--SET
; DELAY COUNT
0028 EB0790 81 JMP INPUT_LOOP
002B C70600001600 R 82 SET 78 US: MOV LOOP_VALUE, 22
0031 8B0E0000 R 83 INPUT_LOOP: MOV CX, LOOP_VALUE
; USE CX TO KEEP TRACK OF DELAY
0035 E480 84 IN AL, SAMPLE_LSB
; SET AL TO LSB OF INPUT SAMPLE
0037 8AD0 85 MOV DL, AL
; PUT THE LSB IN DL (8 BIT XFERS ONLY)
0039 E481 86 IN AL, SAMPLE_MSB
; SET AL TO MSB OF INPUT SAMPLE
; THIS RESTARTS SAMPLE PROCESS
003B 8AF0 88 MOV DH, AL
; PUT AL IN DH TO COMPLETE THE VALUE
003D 83FF0E 89 CMP DI, FIRST_PASS
; WE WANT TO SKIP THE FIRST SAMPLE
0040 7408 90 JZ SKIP_INPUT
0042 83C702 91 ADD DI, SAMPLE_INCREMENT
; INCREMENT DI BY 2
0045 8915 92 MOV DS:[DI], DX
; PUT SAMPLE DATA IN SEGMENT
0047 EB0990 93 JMP DELAY
; AND JUMP TO SOFTWARE DELAY LOOP
004A 83C702 94 SKIP_INPUT: ADD DI, SAMPLE_INCREMENT
; INCREMENT DI BY 2
004D 90 95 NOP
; AND NOP FIVE TIMES FOR EVEN TIMING
004E 90 96 NOP
004F 90 97 NOP
0050 90 98 NOP
0051 90 99 NOP
0052 90 100 DELAY: NOP
; THIS NOP ADDS 3 CLOCKS PER DECREMENT
0053 E0FD 101 LOOPNZ DELAY
; DEC CX AND LOOP--1.5 US PER DECREMENT
0055 81FF0E01 102 CMP DI, SAMPLE_MAX
; COMPARE DI TO SEE IF WE ARE DONE
0059 75D6 103 JNE INPUT_LOOP
; IF NOT, GO BACK FOR ANOTHER SAMPLE
005B 5D 104 POP BP
; OTHERWISE POP BP, DS, AND RETURN
005C 1F 105 POP DS
005D CA0400 106 RET 4H
107 ;

```


Both the RAM and ROM-based configurations will be discussed in this appendix. They are essentially identical processes. In either case, the first step is to define a map of system memory. Once the map is known, the following sequence is suggested for locating code in memory:

- 1) Reserve memory 0H to 03FFH for the Nucleus interrupt vector.
- 2) If the system is RAM based and the code is loaded by the iSBC 957A Monitor, reserve locations 03FFH to 07FFH for the monitor's use.
- 3) Configure each of the necessary portions of the iRMX 86 Operating System and locate them sequentially in memory.
- 4) For a RAM-based development system, allow 2K of RAM for the system Root Job. Placing the Root Job after the portions of the iRMX 86 Operating System, which are relatively fixed in size during development, and before the development code will give the Root Job a fixed address. This will prevent having to move the Root Job and reconfigure the system when the development code grows. For final EPROM-based systems, the Root Job should be placed after the development code.
- 5) Link and locate each of the application code modules sequentially in memory.
- 6) Define the RAM available to the system.
- 7) Define memory NOT available to the system. This includes application code, EPROM, and non-existent memory within the 1 megabyte address space.
- 8) Create the configuration file using the address maps produced by the locate steps and the memory map defined in steps 6 and 7.
- 9) Create the Root Job from this configuration file.
- 10) Load and test the system in RAM.
- 11) If the system has been fully debugged, load the code into EPROM and test the final system.

The above steps are necessary for both the RAM development system and the final EPROM system. Converting this application from RAM to EPROM requires reconfiguring the Nucleus to include only those systems calls required by the application, substituting the Terminal Handler Job for the Debugger Job, removing any remaining system calls to catalog objects for debugging, and remapping the system to the EPROM address space. The memory maps for the development and final application are shown in Figures C-1 and C-2.

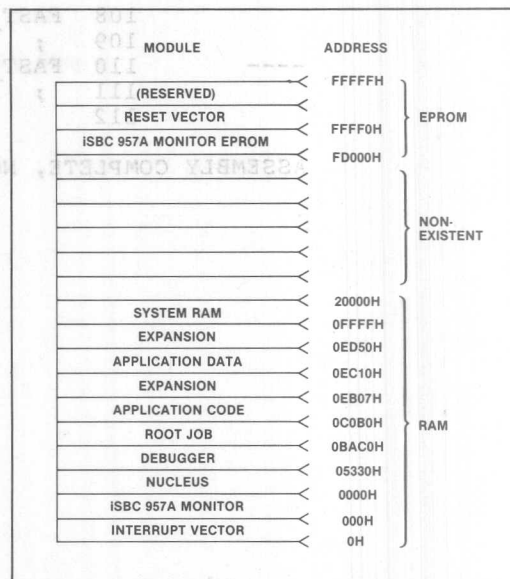


Figure C-1. Development System Memory Map

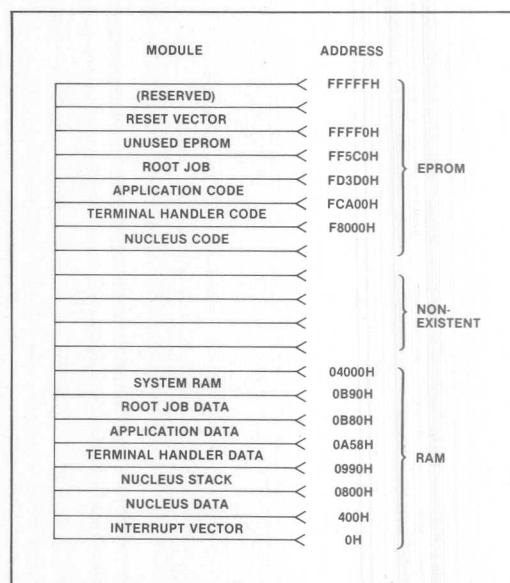


Figure C-2. Final System Memory Map

System configuration is a straightforward but exacting process. As with any such processes, there are some hints that can make development easier. In addition to care in locating the Root Job in memory, users should fix the initialization job entry point and the data RAM addresses.

The Intel PL/M 86 programming language does not allow a procedure to be used until after it has been declared. This requires the initialization procedure to be declared after all the other procedures. Since the initialization is last, changing the other procedures will change the location of the initialization procedure. If the system entry point changes, the system must be reconfigured. The moving entry point can be circumvented by writing a separate initialization task. The Root Job will create only the initialization task which will then initialize the system jobs. The initialization task entry point is fixed by linking it ahead of the other application tasks and by not changing the initialization task during development. The actual system entry points will be bound to the initialization task during linking and locating. The linking and locating steps are a natural consequence of chang-

ing the application code, so binding the fixed system entry point is done automatically during development. The fixed initialization task entry point is used in the configuration file, giving the Root Job an unchanging system entry point.

The remaining moving target during development is the RAM area for data and stack use. If the data and stack RAM is located before or after the application code, with enough extra memory in between for growth during development, the data and stack locations can stay constant. Fixing both the application entry point and the locations of the stack and data segments will allow development of the application code to proceed without requiring frequent reconfigurations.

Single-board microcomputers offer hardware cost-effectiveness for implementing many real-time systems. A compatible, resident, real-time executive program provides savings in software development

An Integral Real-Time Executive For Microcomputers

Kenneth Burgett and Edward F. O'Neil

Intel Corporation
Santa Clara, California

Single-board computers, or microcomputers, that contain central processor, read-write and programmable read-only memory, real-time clock, interrupts, and serial and parallel input/output all on one printed circuit board, have made feasible a whole spectrum of applications which previously could not be economically justified. These microcomputers have also opened up a range of applications where the high functional density of large-scale integration provides advantages over previous solutions such as hardwired logic or relatively expensive minicomputers. While microcomputers readily solve hardware requirements, software for single-board computer applications with real-time characteristics (which are in the majority) has until now been generated individually for each application.

The Intel RMX/80* Real-Time Multi-Tasking Executive simplifies real-time application software development, and at the same time furnishes capabilities optimized for the microcomputer environment. It provides the means to concurrently monitor and control multiple external events that occur asynchronously in real-time. The program framework allows system builders to immediately implement software for their particular applications, and to avoid specific details of system interaction.

Major functions of the executive include system resource access based on task priority, intertask communication, interrupt driven device control, real-time clock control, and interrupt handling. In combination, these functions eliminate the need to implement detailed real-time coordination for specific applications.

Previously, two alternative software approaches were used to solve microcomputer applications. First, many

designers created their own operating executive, individually tailored for each application. Obviously, this approach was expensive and time-consuming. The second approach was to use a minicomputer executive which had been adapted to a microcomputer. Since this software was designed for a different processing environment and then "stripped down," it suffered from major inadequacies when executed on microcomputers. The alternative, RMX/80, has been designed specifically to provide a general-purpose real-time executive tailored to Intel SBC 80 and System 80 microcomputers.

Real-Time System Requirements

All software design approaches for use in real-time applications include capability for concurrence, priority, and synchronization/communication.

Concurrence—Real-time systems monitor and control events which are occurring asynchronously in the physical world. Microcomputer software does not know exactly when external events will occur; however, it must be prepared to perform the necessary processing upon demand, whenever the events actually do occur. Typically, interrupts are used to inform the microcomputer that an event has occurred. At interrupt time, system control software determines what processing to perform, as well as the relative sequence in which processing must take place.

*RMX/80™ is a registered trademark of the Intel Corp, Santa Clara, Calif.

Programs related to external events are processed in an interleaved manner based on interrupt occurrence and priority. For instance, one routine is executing when an interrupt activates, signaling that a higher priority event has occurred. At this point, the routine related to the priority interrupt is started, while execution of the less important routine is discontinued temporarily. When the more important routine is completed, or temporarily halted for some other reason, execution of the less important routine is resumed. In this manner, multiple programs execute concurrently in an interleaved fashion.

Priority—In a real-time environment, certain events require more immediate attention than others because of their significance within the physical world. Immediacy is relative to other processing, and is determined by application requirements. The concept of immediacy or priority, however, is common throughout all real-time microcomputer applications. In priority-based systems, the most important program (one that is not waiting for some physical or logical reason) is the one executing.

A classic illustration of program priority in real-time systems is found in the area of plant control. When the plant begins to fail in a nonrecoverable manner, it is imperative that the plant be shut down as quickly as possible. For this reason, shutdown processing takes priority over all other system demands. Software priority enforces this hardware concept of physical operational events.

Synchronization/Communication—Another common similarity in most real-time systems is the need for synchronization between various events in the physical world which are under microcomputer control. Synchronization is defined as the process whereby one event may cause one or more other events to occur. Communication is the process through which data are sent between input/output (I/O) devices or programs and other programs within the microcomputer system.

An example of the need for synchronization and communication is a microcomputer system for weighing and stamping packages. One part of the system weighs the package, calculates pricing, and releases the package onto a conveyor belt. Price and weight data are communicated to another part of the system which stamps the data onto the package after it arrives at a sensor station. Synchronization is demonstrated by the occurrence of one event—package arrival—causing another event—package stamping—to occur.

Compatible Benefits

To satisfy real-time microcomputer software requirements, the RMX/80 Real-Time Executive software (Fig 1) was designed. This program differs from existing software systems by offering capabilities directly related to the single-board microcomputer environment in which it operates. These capabilities have two major bottom-line benefits compared with equivalent minicomputer systems. First, the executive code is compact enough to allow a large number of real-time applications to be processed on a single microcomputer board. To accomplish this capability, its nucleus is optimized to reside in less than 2k bytes [ie, in a single 16k programable read-only memory (p/ROM)], thereby allowing up to 10K of onboard memory for application-related software and storage.

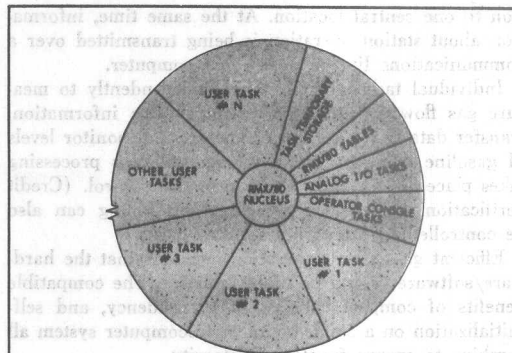


Fig 1 A typical RMX/80 system. Multiple tasks control a given application. Nucleus controls execution of both user and executive tasks through task-to-task communication, real-time clock, priority resolution, and interrupt handling facilities. All tasks within an RMX/80-based application use at least some of these capabilities; other optional executive tasks include debugger, free-space manager, and device control for operator's console, diskette file system, analog subsystems, and high speed mathematics unit.

Second, the executive may be p/ROM-resident. When the microcomputer system is powered on, the software system (executive plus application programs) is automatically initialized and begins execution of the highest priority application task. Typical major real-time executives, however, are totally random-access read-write semiconductor memory (RAM)-resident, which means they must be initialized (booted) from a peripheral device, such as diskette, cassette, or communications line, into microcomputer memory. The need for peripheral devices significantly increases the total cost of traditional real-time executive-based solutions.

Sample Application

Functioning as a real-time executive for microcomputers, this software system provides facilities for orderly control and monitoring of asynchronously occurring external events. Although these events may differ widely from application to application, facilities are adaptable to nearly all processes where the microcomputers are used, including process and machine control, test and measurement, data communications, and specialized on-line data processing applications (where one or more terminals access diskette-based data). The executive is particularly useful in dedicated low cost applications which were not economically feasible before the advent of microcomputers. For example, consider the requirement of gas pump control in a service station (Fig 2).

In this station, a microcomputer system operating with RMX/80 concurrently monitors and controls multiple gas pumps, and sends price and volume informa-

tion to one central location. At the same time, information about station operation is being transmitted over a communications line to a regional computer.

Individual tasks are developed independently to measure gas flow, calculate and display price information, transfer data to the central computer, and monitor levels of gasoline in underground storage. All this processing takes place concurrently under program control. (Credit verification, charge slip printing, and billing can also be controlled by additional software tasks.)

Efficient gas station operation demands that the hardware/software system be highly reliable. The compatible benefits of compact code, p/ROM residency, and self-initialization on a single-board microcomputer system all combine to ensure functional integrity.

Software Structure

RMX/80 simplifies the effort for developing a real-time system, first, by providing many commonly required software functions. Second, its software structure promotes efficient program development. Programmers who are familiar with structured programming will find task orientation both natural and easy to use.

Tasking means that a larger program is divided into a number of smaller, logically independent programs or tasks. The key is to identify functions that may occur concurrently. For example, consider the tasks required for a terminal handler—real-time asynchronous I/O between an operator's CRT terminal and the executive.

Input Handler Task—One task must be ready to accept a data character from the terminal at any time. This is done by responding to an interrupt signal from the terminal and then accepting the data character. The task immediately passes the input character to a subsequent task automatically and then goes back to wait for another interrupt.

Line Buffer Task—As characters are received from the input handler they must be placed into a buffer to form a line. Eventually, the buffer will be filled or the logical end-of-line will be signaled by a carriage return character. At this point, the line buffer must be sent to some other task for processing.

Echo Driver Task—For a full-duplex terminal, it is necessary to return each input character to the terminal for display on the CRT screen. This task waits for a character, which could be sent by either the line buffer or input handler task, and then sends the character to the terminal. It then waits for the next character.

Note that input handler and echo driver are described as waiting for an event. Within the RMX/80, that is literally the case. While they wait, however, system resources are available for other tasks, such as that of the line buffer. Thus, effective processing may occur concurrently with necessary waiting periods. Notice also that a number of other tasks may also be active within the system. In fact, the greater the number of tasks running concurrently, the more effectively system resources are used. Concurrent operation eliminates many time wasting procedures from a real-time system. For example, the executive can eliminate the need for many timing loops where the processor simply executes a no-operation instruction repeatedly while waiting for an event to occur.

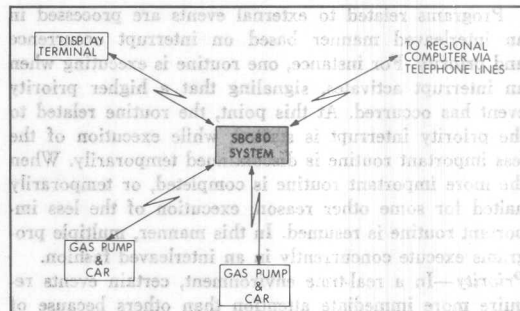


Fig 2 Microcomputer control for gas pump automation. In this example, executive-based system simultaneously controls two pumps, displays information on operator's console, and communicates with regional computer. At a given time, more or fewer functions could be operating concurrently. System expansion can be easily accomplished by adding tasks and modular hardware.

Within the executive, tasks not only are logically independent, they are also physically independent, actually contending with each other for the use of the processor and other system resources. The executive resolves this contention based on the priority of each task.

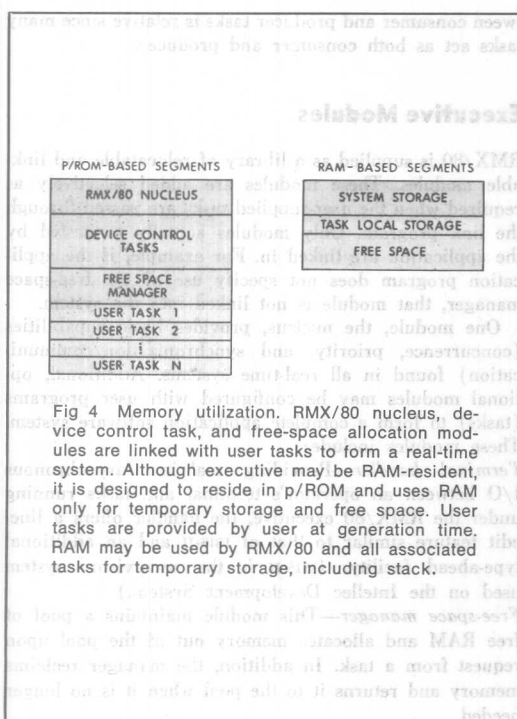
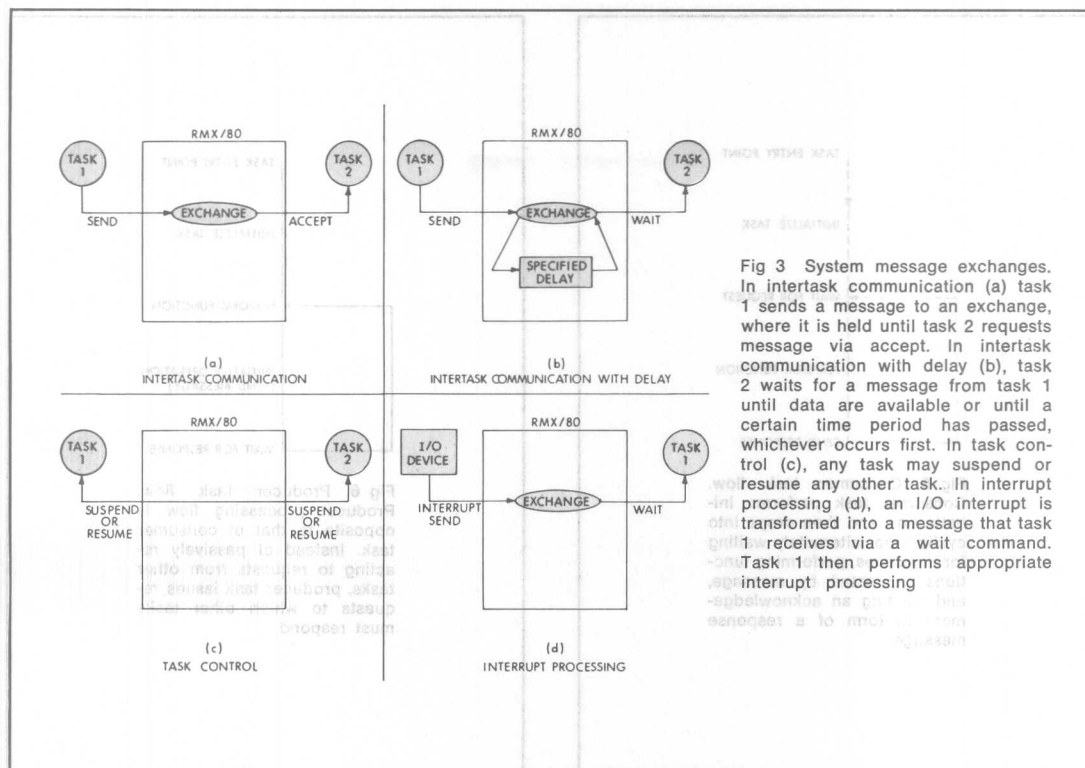
In the terminal handler example, it is clear that the input handler must have highest priority, since acceptable performance cannot tolerate the loss of data. Second highest priority is given to the echo driver, so that data appearing on the screen remain coordinated with the input. Lowest priority goes to the line buffer, since that function does not depend directly on an external asynchronous event. There are no particular real-time constraints on the line buffer as long as the input characters are eventually processed.

It is possible to write the entire terminal handler as a single large task instead of as several smaller tasks. However, consideration must be given other high priority tasks operating within the system which may not be able to gain control while a low priority portion of the terminal handler, such as the line buffer task, is executing. Therefore, tasks assigned as high priority are generally kept as short as possible. If the terminal handler were written as one large task, it could tie up the entire processing system for a relatively trivial function.

Task States

Two task states have been implied—running and waiting. A running task is always the task which currently has the highest priority and is not suspended or waiting. A waiting task remains in the wait state until it receives a message or an interrupt for which it is waiting or until a specified time period has passed. The wait period can be timed using the system clock.

A running task may suspend itself on some other task in the system. A suspended task cannot begin execution again until some running task orders it to resume. As an example, a password routine might temporarily suspend the echo driver of the terminal handler so that the password is not displayed. (The password routine must



remove the password from the line buffer, or it will be displayed as soon as execution of the echo driver is resumed.)

A task may also be in the ready state. A ready task is one that would be running except that a task with higher priority temporarily controls the system resources. The executive maintains a list of all tasks that are ready to run. The next task to be run is always the task with the highest priority in the ready list.

The running task relinquishes its control of the system by

- (1) Putting itself into a wait state
- (2) Suspending itself
- (3) Sending a message to a higher priority task, which if it has the highest current priority, becomes the running task
- (4) Being preempted by an interrupt to a higher priority task

In the case of an interrupt, the executive saves the status (contents of registers, etc) of the interrupted task so that it will be restarted correctly.

Message Exchanges

Tasks communicate with each other by sending messages (Fig 3). The sending task constructs the message to be sent in RAM or uses a previously assembled message!

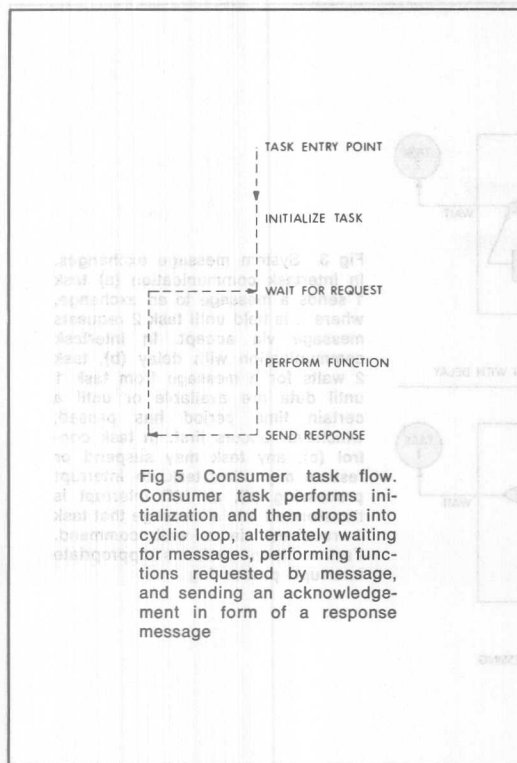


Fig 5 Consumer task flow. Consumer task performs initialization and then drops into cyclic loop, alternately waiting for messages, performing functions requested by message, and sending an acknowledgement in form of a response message

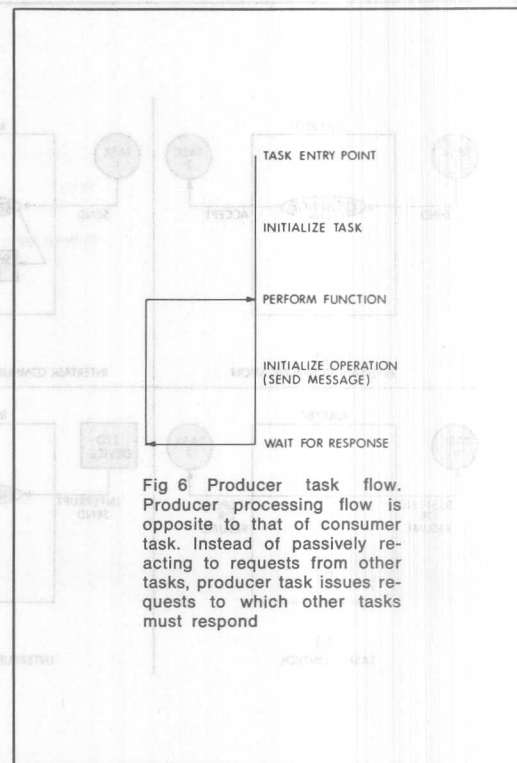


Fig 6 Producer task flow. Producer processing flow is opposite to that of consumer task. Instead of passively reacting to requests from other tasks, producer task issues requests to which other tasks must respond

The sending task then issues a SEND command that posts the address of the message at an exchange.

An exchange is simply a set of lists maintained by the executive. The first list contains the addresses of messages available at that exchange. The second list consists of a list of tasks that are waiting for messages at that exchange. When a task enters a wait state, it specifies the exchange where it expects eventually to find a message. The task may wait indefinitely, or it may specify that it will only wait a specific period of time before resuming execution.

Messages, together with the exchange mechanism, provide for automatic intertask communication and also for task synchronization. For example, a message to a particular task may specify that the task is to send a response to a certain exchange. Thus, the original task may request an acknowledgement response to its message, or it may specify that a message is to be sent to a third task. RMX/80 treats interrupts like messages, the only difference being that interrupts have their own set of exchanges.

Note that the sending and receiving of messages classifies tasks into two types—message consumers and message producers. A consumer task waits for a message, performs an action based on the message, and then returns to the wait state until another message is received. A producer task initiates its function by sending a message to another task, waits for a response, and then sends another message. Figs 5 and 6 graphically illustrate the processing within these two tasks. The distinction be-

tween consumer and producer tasks is relative since many tasks act as both consumer and producer.

Executive Modules

RMX/80 is supplied as a library of relocatable and linkable modules. These modules are added selectively as required when the user-supplied tasks are passed through the link program. Only modules actually requested by the application are linked in. For example, if the application program does not specify use of the free-space manager, that module is not linked into the system.

One module, the nucleus, provides basic capabilities (concurrency, priority, and synchronization/communication) found in all real-time systems. Additional, optional modules may be configured with user programs (tasks) to form a complete application software system. These modules include:

Terminal handler—Providing real-time asynchronous I/O between an operator's terminal and tasks running under the RMX/80 executive, the handler offers a line-edit feature similar to that of ISIS-II and an additional type-ahead facility. (ISIS-II is the supervisory system used on the Intellec Development System.)

Free-space manager—This module maintains a pool of free RAM and allocates memory out of the pool upon request from a task. In addition, the manager reclaims memory and returns it to the pool when it is no longer needed.

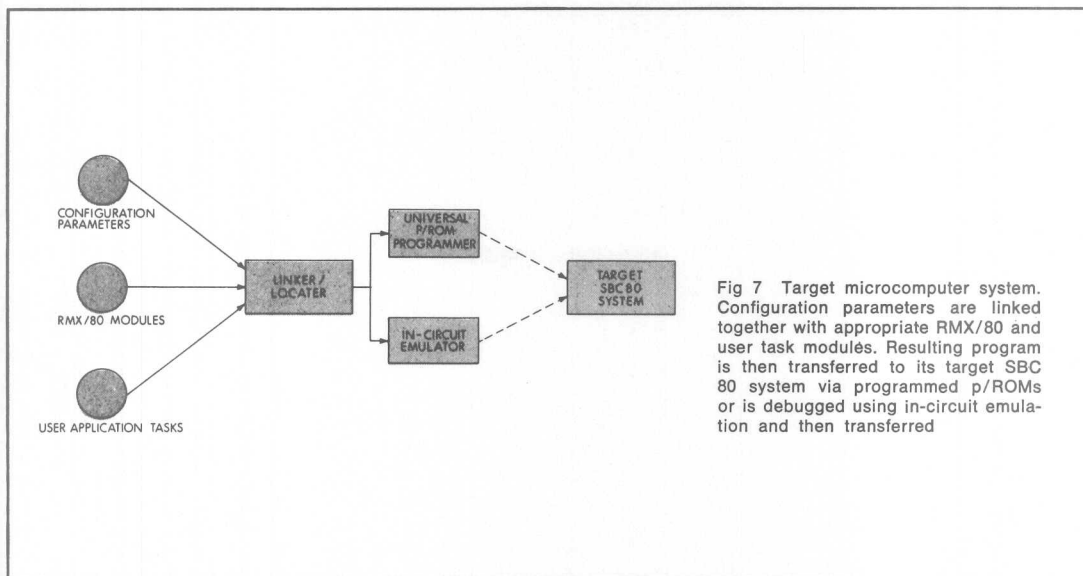


Fig 7 Target microcomputer system. Configuration parameters are linked together with appropriate RMX/80 and user task modules. Resulting program is then transferred to its target SBC 80 system via programmed p/ROMs or is debugged using in-circuit emulation and then transferred

Debugger—Designed specifically for debugging software running under the RMX/80 executive, the debugger is used by linking it to an application program or task. Thus, it can be run directly from the single-board computer's memory. In addition, an in-circuit emulator, such as ICE-80, can be used to load and execute the debugger, providing all resources of the Intellec development system to simplify debugging effort.

Analog interface handlers—Consisting of RMX/80 tasks, these handlers provide real-time control for SBC 711, 724, and 732 systems.

Diskette file systems—Giving RMX/80 users diskette file management capabilities, the diskette driver allows users to load tasks into the system and to create, access, and delete files in a real-time environment without disrupting normal processing. All file formats are compatible with ISIS-II for both single and double density systems.

In addition to application program module or task requirements, the user also supplies a set of generation parameters. These parameters are a set of tables that inform the executive of the number of tasks and exchanges in the system. Fig 7 illustrates the system generation process.

Summary

The significance of RMX/80 to software design parallels the significance of the single-board computer to hardware design. Microcomputers allow designers without extensive experience in digital systems to bring computer processing power into their applications. Similarly, the executive relieves the hardware designer of much software design required for real-time applications. Designed to facilitate growth, since new software needed to support hardware expansions can be supported easily by the addition of new tasks, this executive also substantially re-

duces recurring costs because it requires a minimum of memory and does not require peripheral bootstrap loading devices. RMX/80 results in economical, shorter, and more flexible software development efforts when designing, building, and verifying real-time user applications.

Bibliography

- C. G. Bell, A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971
- P. Brinch-Hansen, *Operating Systems Principles*, Prentice Hall, 1973
- E. W. Dijkstra, "The Structure of the *THE* Multiprogramming Systems," *Communications of the ACM*, May 1968, pp 341-346
- E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, Mass, 1972
- D. M. Richie, K. Thompson, "The UNIX Time Sharing System," *Communications of the ACM*, July 1974, pp 135-143

Kenneth Burgett, currently software project leader for OEM products and project leader for RMX/80 at Intel Corp, has been involved in system programming for a variety of computers. He holds a BS degree in mechanical engineering from Marquette University.

Edward O'Neil received a BS degree from Louisiana State University and an MBA degree from California State University. Currently software marketing manager for the single-board computer family at Intel Corp, his background includes design and development of real-time operating systems.

Introducing the RMX/86™ realtime, multitasking, 16-bit operating system



As we have seen, the evolution in which electronic intelligence is spreading to more and more places, and in which the application sophistication at each of these places is growing as well. For a man to do what it takes to program all the new microcomputer-based systems, he must be able to calculate what that means in terms of how many computer programs he can run at once. By 1990, we come out with a requirement for over one million computer programs to be run at once.

August 1980

Such systems when needed in OEM microcomputer applications, for problem-solving and for OEM afford many years of use to develop the intimate familiarity with the hardware that is needed to design executive software. But with Intel's RMX/86 system he won't have to.

In addition, this second-generation 16-bit system adds error handling flexible communication devices and other advanced OS capabilities to provide operators available on the mature RMX/86 package.

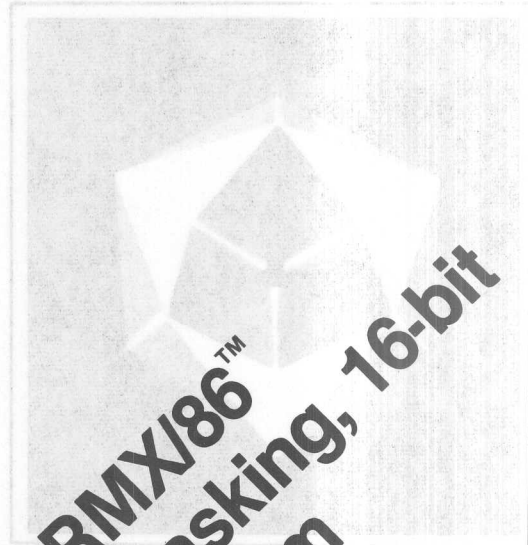
All real-time multitasking systems require executive software to manage the resources shared by the programs (CPU time, memory and I/O) against to registers and then allocate the resources according to established priorities. Normally, these functions are intermingled in an OS with higher level system functions that often prove significant in microcomputer applications.

The RMX/86 package contains all these required executive functions in a single module, called the nucleus. Other modules in the package join the executive system to its applications by adding higher-level operating system functions such as disk file systems. The application programs and optional modules connect to the nucleus with simple software interfaces.

Since the nucleus is necessarily organized in services the software foundation for planning high-level operating system functions and services is of high-level programs. Although not a microcomputer application, the dedicated, ready-to-run high-level functions

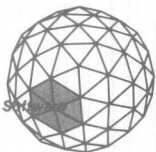
The OEM way

The RMX/86 system software fits in with the OEM way of doing business and makes it easy to apply the RMX/86 system to a wide variety of applications. The RMX/86 system is designed to be used in the lowest level of the system, where the hardware is most likely to be found. The RMX/86 system is designed to be used in the lowest level of the system, where the hardware is most likely to be found.



Introducing the RMX/86™ Realtime, Multitasking, 16-bit Operating System

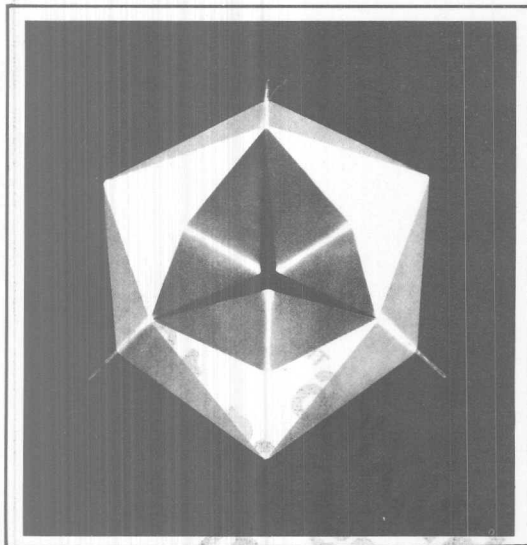
Preview Magazine, May/June 1980



PRODUCT ANNOUNCEMENT

Introducing the RMX/86™ realtime, multitasking, 16-bit operating system

"Now we have a situation in which electronic intelligence is spreading to more and more places, and in addition, the application sophistication at each of these places is growing as well. So, when we multiply what it takes to program all the new microcomputer products so that they can be applied, and calculate what that means in terms of how many computer programmers will be needed by 1990, we come out with a requirement for over one million computer programmers!" Excerpted from a speech by Andrew S. Grove, President, Intel Corporation, given before the New York Society of Security Analysts, January 24, 1980.



As production technologies such as VLSI (Very Large Scale Integration) proliferate—with their greater levels of density—the use of low-cost microcomputer hardware to solve increasingly complex application problems has created a crisis in software. A partial solution to the programmer shortage would be the creation of "building blocks" for system development. It is this modular, building block concept upon which the RMX/86 Operating System is constructed.

The RMX/86™ operating system

This modular operating system for Intel 16-bit microcomputers allows custom operating systems to be assembled largely from off-the-shelf software com-

ponents. Such systems when needed in OEM microcomputer applications, pose problems—rarely can an OEM afford man-years of effort to develop the intimate familiarity with the hardware that's needed to design executive software. But with Intel's RMX/86 system he won't have to.

In addition, this second-generation 16-bit system adds error handling flexible command-line decode, and other advanced OS capabilities to previous options available on the mature RMX/80 package.

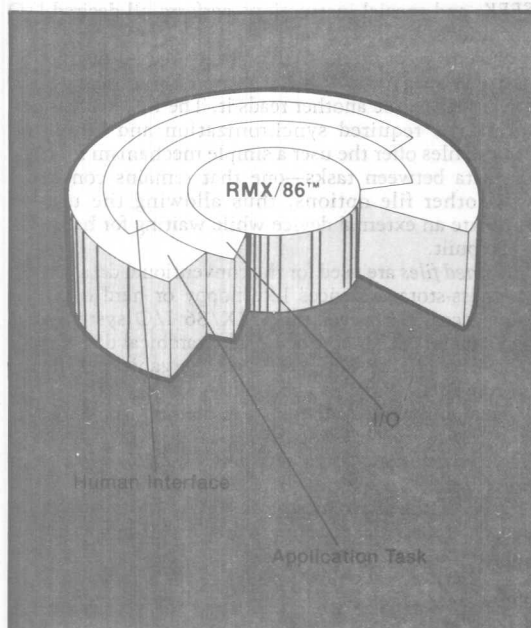
All real-time multitasking systems require executive software to manage the resources shared by the programs (CPU time, memory and I/O), respond to interrupts and then allocate the resources according to established priorities. Normally, these functions are intermingled in an OS with higher-level system functions that often prove superfluous in microcomputer applications.

The RMX/86 package combines all those required executive functions in a single module, called the nucleus. Other modules in the package tailor the executive system to its application by adding higher-level operating system functions such as disk-file systems. The application programs and optional modules connect to the nucleus with simple software interfaces.

Since the nucleus is essentially open-ended, it serves as the software foundation for expanding both higher-level operating system functions and varieties of application programs. Although most microcomputer applications are dedicated, many do require such higher-level capabilities.

The OEM way

The modular approach to system software fits in with the way most OEMs (and high-volume end users) apply single-board computers. They generally start with a minimum amount of hardware marketed at the lowest cost possible. Then, when their users are fully utilizing the original functions and are willing to buy more, the



The RMX/86 Nucleus is the heart of the operating system and is surrounded by application, human interface and input/output facilities.

OEM adds functionality by increasing the executive facilities, application tasks and hardware—allowing the fastest turnaround possible.

To see how the RMX/86 operating system works, consider a typical example. Say an OEM develops a factory heating and air-conditioning control system with a single-board computer, analog I/O, special control devices and operator terminal. He uses the nucleus, and terminal handler modules, plus user tasks stored in EPROM.

But suppose the OEM's customer wants to enhance the system with, say, disk storage. He simply adds a disk controller board and I/O system software. The original programs could also be made disk-resident. If the original application had been based on a conventional executive, extensive redevelopment would have been necessary.

If, on the other hand, the application had used the full I/O system from the start, initial sales would have suffered because the OEM's system would have been more costly. And if the software is not extendible, future sales are lost.

The historic problem with conventional "general-purpose" operating systems is that they usually depend on hardware that the application would not otherwise need—for example, large disk systems while a typical OEM system uses no mass storage at all. Modular operating systems are designed to accommodate a diversity of applications without undue hardware imposed on them.

For an even closer fit with customers applications,

the RMX/86 modules closely parallel hardware modularity.

Corresponding to the hardware, the RMX/86 nucleus manages CPU, memory and I/O sources. When linked to optional modules it can support standard iSBC devices like consoles and disk controllers. Similarly, users may add device driver software modules for their custom peripherals. All software can reside in EPROM/ROM if mass storage is not available.

The RMX/86 operating system is designed for configuration, to user specification, on an Intellec development system. The same system supports application software development as well as linking and locating both high-level and assembly language.

What does a nucleus do?

A real-time multitasking nucleus gives a program the means to monitor and control external events. Tasks running concurrently, using the services of the executive are signaled with interrupts and the executive schedules resources for the tasks on the basis of priority. For example, a task trying to bring an oven's temperature under control should have a much higher priority than a report generating task. The nucleus' scheduler decides whether a running task should be interrupted to process data from an interrupting device.

The RMX/86 nucleus makes event-driven priority scheduling possible by monitoring system states, determining task requirements, allocating the resources and gathering the resources for reallocation after use has completed. Resources include CPU time, memory and I/O.

As in most priority based systems, the highest-priority task that's ready to run uses the CPU; others are put on a ready list. After servicing an interrupt, the nucleus returns the CPU to the highest-priority ready task.

Managing memory space

A memory manager, built into the nucleus, handles the 8086's megabyte addressing range, insuring that memory is used efficiently. The memory manager organizes memory into a tree-structured hierarchy of pools according to each job's requirements, and provides for allocation and deallocation of memory from these pools. When the nucleus or application tasks require memory, the memory manager allocates it on the basis of segments that are multiples of 16-bytes (to 65,536 bytes) corresponding to the 8086's segmented memory architecture.

The RMX/86 nucleus provides three sets of facilities for tasks to communicate with one another. Each of the three is optimized for either data transfer, synchronization or mutual exclusion.

Mailboxes are provided to allow tasks to send data, in the form of segments, to one another. Since each seg-

ment is referenced by a description, of course, only pointers are moved rather than massive blocks of data.

Semaphores offer low overhead mechanisms for synchronization operations. For example, one task can simply set a semaphore to tell another task that an event has happened ("Analog data received").

The third communication facility offers a unique mutual exclusion facility. Regions are defined as a body of code which manipulates a shared resource. The RMX/86 nucleus insures that while one task is manipulating the resource others don't inadvertently affect it.

All intertask communication functions are accessible with simple calls to the nucleus. For example, to access a mailbox the task simply names the mailbox desired. The programmer has no need to know the internal structure of the mailbox, since all functions are accessible through easily programmed interfaces.

Fault tolerance

Another RMX/86 feature, exceptional-condition handling makes error handling simple rather than elusive. Programs can be designed to manage error conditions and take the appropriate corrective action.

First, there's an option to specify to the nucleus whether or not there should be any error handling for a particular task. A programmer can use a default handler to abort a task or he may program a specific course of action—for example, load a fresh copy of the program and try again.

The exception-condition facilities also detect programming errors such as a wrong call to the nucleus and system problems, like insufficient memory. All-in-all the OEM will find that fault tolerance is no longer just a buzzword.

Beyond the microprocessor

Most microcomputers are used with a range of peripheral devices, from serial lines to mass storage.

The RMX/86 I/O system provides the user with a very general file concept; that of files as a data sink or source. The characteristics of specific storage medium dictate the access techniques for a given file. For example, a disk file may be accessed either in sequential or random fashion, while a file accessed over a serial link (USART) must be processed serially.

Using the data sink/source concept, the user can develop application programs without worrying about the physical characteristics of the device.

Such device independence simplifies application programming and allows programs to be used with a variety of devices.

The RMX/86 I/O system supports three types of logical files:

Physical files represent the lowest interface level which retains its device-independent characteristics. Physical files provide a simple, consistent interface to all device drivers. OPEN, CLOSE, READ, WRITE,

SEEK, and special instructions perform all desired I/O operations.

Stream files provide a temporary data-transmission path between tasks. One task may write data to the stream file while another reads it. The I/O system performs the required synchronization and buffering. Stream files offer the user a simple mechanism for passing data between tasks—one that remains consistent with other file options, thus allowing the user to simulate an external device while waiting for hardware to be built.

Named files are used for the conventional data storage on mass-storage devices like floppy or hard disks for later access. However the RMX/86 I/O system goes one step further by setting up a hierarchical directory of files. This feature allows the user to organize his files to be consistent with his application.

The named-file system has another advantage: It permits file-access checking, so a user can decide which of his files he wants to protect and which to share with other users.

Each of the file options can be configured independently; so that the user may select only the features he needs—nothing more.

Furthermore, the RMX/86 I/O system is designed to allow the user to easily add his own device drivers to the system as the need arises.

The RMX/86 operating system was designed to offer the user a wide spectrum of convenient human interface functions. For example, the user of mass-storage systems is provided display directory and copy file utilities. Additionally, the OEM who needs his own interactive capability can either use the standard RMX/86 facilities or easily extend the system's human interface routines to meet his specific application requirements.

Summary

The software crisis can be overcome only by introducing a broad base of system software functionality into the market, allowing OEMs to concentrate on their area of expertise.

Intel's RMX/86 operating system takes a major stride forward in providing extendible, proven tools for OEMs to use in creating custom operating systems for their application.

The RMX/86 operating system thus provides the OEM with a powerful new system building block, enhancing the productivity of system generation. New dynamic tools like the RMX/86 operating system allow users to deliver their product into the marketplace much earlier, thereby capturing an important competitive advantage.



Modular multitasking executive cuts cost of 16-bit-OS design

A modular, real-time multitasking operating system for single-board computers allows custom operating systems to be assembled largely from off-the-shelf software components. Such systems, when needed in OEM single-board μ C applications, pose problems—rarely can an OEM afford man-years of effort to develop the intimate familiarity with the hardware that's needed to design executive software. But with Intel's RMX/86 system for iSBC 86 single-board computers, he won't have to.

In addition, this second-generation, 16-bit system added error-handling, flexible command-line decode, and other advanced OS capabilities to previous options available on the older RMX/80.

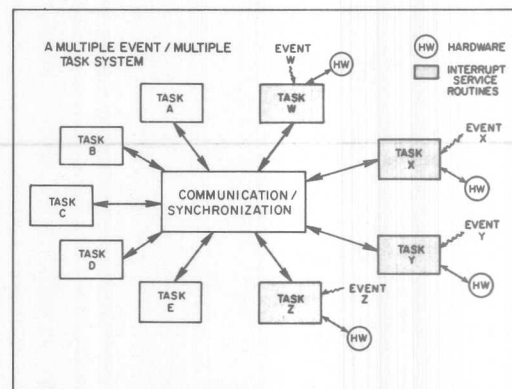
All real-time multitasking systems require executive software not only to manage the resources shared by the task programs (CPU time, memory and I/O), but also respond to interrupts and then allocate the resources according to established priorities. Normally, these functions are intermingled in an OS with higher-level system functions that often prove superfluous in single-board computer applications.

The RMX/86 OS package combines all those required executive functions in a single module, called the nucleus. Other modules in the package tailor the executive system to its application by adding higher-level OS functions such as disk-file systems. The task programs and optional modules connect to the nucleus with simple software interfaces. Users can also add their own extensions.

Since the nucleus is essentially open-ended, it can serve as the software foundation for expanding both the operating system and the variety of task programs. Although most single-board computer applications are dedicated, many do require higher-level capabilities.

The OEM way

The modular approach to system software fits in with the way most OEMs (and high-volume end users) apply single-board computers. They generally start with a minimum amount of hardware (often, just a



1. In a real-time multitasking system, task modules (A through E) can often perform their functions only after hardware-generated interrupts are serviced (highlighted). The executive in such a system provides intertask communications and synchronization.

RMX/80 vs RMX/86 features

RMX/80	RMX/86
Nuclei	Nuclei
For iSBC 80/10	One serves all iSBC boards
For iSBC 80/20	Free-space manager
For iSBC 80/30	Exceptional-conditions handler
Optional modules	Optional modules
Disk-file system	I/O system
Disk I/O	Hierarchical file system
Terminal handlers	Numbered file system
Free space manager	Internal file system
Analog I/O handlers	Physical file system
Bootstrap loader	Interfaces for custom files and I/O
Debuggers	Human interface system
Support packages	Command-line decoder
Fortran-80 run-time	
Basic-80 interpreter	
8080/8085 fundamental support	

Joseph Harakal, Software Product Manager, Intel Corp., 5200 Elam Young Pkwy., Hillsboro, OR 97123.

single board) to begin an application at low cost. Then, when users are satisfied with the original functions and are willing to buy more, the OEM adds the executive options, tasks and hardware—with as fast a turnaround as possible.

The problem with conventional "general-purpose" software systems is that they usually depend on hardware that the application may not need—for example, standard peripherals whereas a typical OEM system uses special peripherals. Modular OSs are designed to accommodate both.

What's more, a conventional system can make it difficult and/or awkward to use new peripherals or new technology, such as magnetic-bubble memory—most command-line decoders, for instance, are not accessible to the user. So, a user may discover that there is no straightforward way of adding new facilities.

Call it foundation software

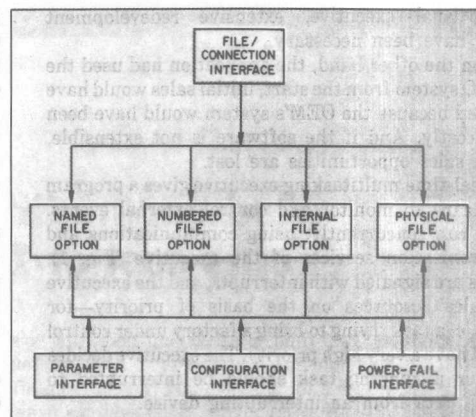
For an even closer fit with single-board computer applications, the RMX/86 modules closely parallel hardware modularity: Each computer board contains program and data memory, serial and parallel I/O, and other generally required functions in addition to the CPU. Each user's system is expandable with optional modules. Frequently used devices like disk controllers and analog I/O are available. In addition, the user can connect custom devices to his system via the Multibus architecture.

Corresponding to the hardware, systems software manages CPU, memory and I/O resources. Linked to optional modules, it can support standard ISBC devices like consoles and disk controllers. Similarly, users may add device-driver software modules for their custom peripherals. All software can reside in EPROM/ROM if mass storage is not available; otherwise, most of the system can be disk-resident. The disk-file module is suitable for such applications as data logging.

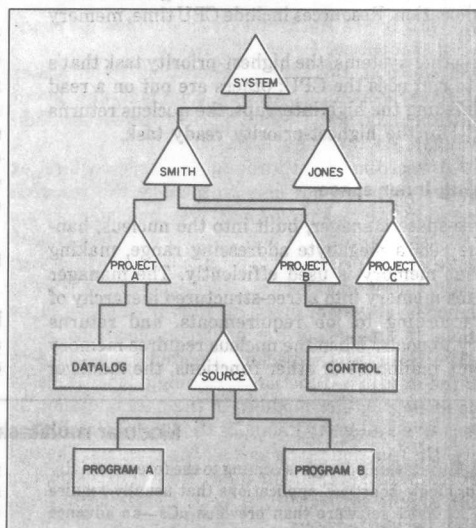
The RMX/86 is designed for configuration on an Intel development system according to user requirements. The same system supports task-module development as well as linking and locating in both high-level and assembly languages. It also provides libraries of frequently used program functions to minimize the amount of code the system designer must write and debug.

To see how the RMX/86 works, consider a typical example. Say an OEM develops a factory heating and air-conditioning control system having a single-board computer, analog I/O, special control devices and an operator terminal. He uses the nucleus, analog-handler and terminal-handler modules, plus user tasks stored in EPROM.

But suppose the OEM's customer wants to enhance the system with, say, disk storage. He simply adds a disk controller board and uses I/O system software. The original programs could also be made disk-resident. If the original application had been based on a



2. The RMX/86 operating system treats all I/O as files—a feature that makes it easy to add new peripherals and special files.



3. A hierarchical file system eliminates the need for scanning all the files on a disk. If Smith is working on Project A, he only has to choose between the files related to his project.

conventional executive, extensive redevelopment would have been necessary.

If, on the other hand, the application had used the full I/O system from the start, initial sales would have suffered because the OEM's system would have been more costly. And if the software is not extensible, future sales opportunities are lost.

A real-time multitasking executive gives a program the means to monitor and control external events. Tasks run concurrently, using communications and synchronization services of the executive (Fig. 1). Events are signaled with interrupts, and the executive schedules resources on the basis of priority—for example, a task trying to bring a factory under control should have a very high priority. The executive decides whether a running task should be interrupted to process data from an interrupting device.

The RMX/86 nucleus makes such event-driven priority scheduling happen through resource management. It monitors system states, determines task requirements, allocates resources and gathers them for reallocation. Resources include CPU time, memory and I/O.

As in other systems, the highest-priority task that's ready to run uses the CPU; others are put on a read list. Finishing the high interrupt, the nucleus returns the CPU to the highest-priority ready task.

Managing inner space

A free-space manager, built into the nucleus, handles the 8086's megabyte addressing range, making sure that memory is used efficiently. The manager organizes memory into a tree-structured hierarchy of pools according to job requirements, and returns memory to pools. When the nucleus requires memory for a job, mailboxes or other functions, the manager

provides it. Or, when a task requests memory—for example, to input data—the manager allocates it. This is done in segments that are multiples of 16 bytes, corresponding to the 8086's segmented memory.

Tasks send data to each other through mailboxes which contain messages located in RAM. As in other systems, separate mailboxes are used for receiving and for responding.

In the RMX/86, however, mailboxes provide several options, including synchronization, mutual exclusion (which prevents one task from destroying another task's data) and communications—for example, with the outside world—through modules such as the terminal handler.

The RMX/86 nucleus also provides other means for communications. One example is semaphores—low-overhead mechanisms for synchronization operations, resource allocation and mutual exclusion that require a simple flag. For example, one task can simply set a flag to tell another task that an event has happened ("Analog data received").

All these functions are accessible with simple calls to the nucleus. For example, just two calls are needed to use the free-space manager: CREATE-SEGMENT to request memory and DELETE-SEGMENT to relinquish memory. Likewise, to obtain an mailbox, the task simply names the mailbox desired. The programmer doesn't need to know the internal structure of the executive, since the functions are accessible through easily programmed interfaces.

Errors, big and small

Another RMX/86 feature, the exceptional-condition handling, makes error handling selective rather than all or nothing. Programs can be designed to manage error conditions and take corrective action.

Modular multitasking comes on

Multitasking design is coming to the fore, especially for single-board- μ C applications that usually require a lot more software than previous μ Cs—an advance from simple foreground-background programs to techniques based on event priorities.

Although single-board μ Cs started out in the mid-1970s at the low end of the OEM performance range, they have now reached the top in performance and memory capacity. As more and more OEMs and users take advantage of that increased capability, the size of applications programs grows—and grows.

In the same period, the costs for developing software, salaries and overhead have almost doubled. Moreover, skilled programmers have become one of the industry's most limited resources. No wonder that software costs comprise up to 80% of system development costs today, and that the emphasis has shifted from in-house software design to buying off-the-shelf programs.

Because multitasking designs have to be highly modular, time-saving tools such as high-level lan-

guages, program libraries and off-the-shelf software can be used freely to help keep development, maintenance and expansion costs under control. Code written in high-level languages is a bargain today, compared to code written in assembly language: around \$2.50 a byte vs \$10 for assembly language. High-level code is not as compact, but it's far more cost-effective for the 80% of the tasks that run only about 20% of the time in typical applications.

Today, there's a growing choice of languages. Structured languages like PL/M and Pascal fit well into the "top-down" modular design technique used to divide an application into tasks. Others, like Fortran for mathematical applications and Basic for easy end-user programming, are also available.

In general, a real-time multitasking executive offers a reasonable choice for the user who has a lot of software to write, must meet special requirements, and has no time to develop a custom operating system. The RMX/86 system, with its modular design, fills the bill to save development cost.

First, there's an option to specify to the nucleus whether or not there should be any error handling for a particular task. A programmer can write his own handler either to abort a task or to program a specific course of action—for example, report exceptional conditions and continue with next instruction; load copy of module and try again; start alarm program.

The exceptional-conditions support also detects programming errors such as a wrong call to the nucleus and system problems like insufficient memory. Naturally, the RMX/86 provides all normal OS functions. System options include a terminal handler for CRT and TTY consoles device drivers for Intel's floppy and hard-disk controllers and an integrated I/O system. A subsystem of the I/O system supports tree-structured directories hierarchical named files (Fig. 2).

I/O features are vital

Most single-board computers are used with special peripheral devices, and many with other kinds of files and media. So, the I/O system is designed to make it easier to add special files, new peripherals and custom device drivers—the user need never feel locked in.

The RMX/86 I/O system provides the user with a very general file concept—as a data sink or source.

The characteristics of a specific storage medium dictate the access techniques for a given file. For example, a disk file may be accessed either in sequential or random fashion, while a file accessed over a serial link (USART) must be processed serially.

Using the data sink/source concept, the user can develop application programs without worrying about the physical device where the data will be stored. Such device independence simplifies application programming, and existing programs can be used with many devices.

The RMX/86 I/O system supports three types of files.

Physical files represent the lowest interface level to retain device-independent characteristics. They provide a simple, consistent interface to all device drivers. OPEN, CLOSE, READ, WRITE, SEEK and special instructions perform all desired I/O operations.

Stream files provide a temporary data-transmission path between tasks. One task may write data to the stream file while another reads them. The I/O system performs the required synchronization and buffering. Stream files offer the user a simple mechanism for passing data between tasks—one that remains consistent with other file options. The user can simulate an external device while waiting for hardware to be built.

Named files are used for the conventional data storage on mass-storage devices like floppy or hard

```

1      TASKB1: PROCEDURE PUBLIC;
2
2      call rgend$inittask;
2      CALL INIT;
2      do forever;
3
3          msg$token=r$receivemessage(mailbox$,
-      0FFFFH,@resp$ex,@ex$val);
3          call r$send$message(r$normal$th$out,
-      msg$token,out$resp,@ex$val);
3          msg$token= r$receivemessage(out$resp,
-      ,0FFFFH,@resp$ex,@ex$val);
3          call r$delete$segment(msg$token,@ex$val
-      al);
3
3          end; /* of do forever */
2      end;
```

4. RMX/86 can easily be expanded with user-coded tasks. The one shown here initializes a user program by calling the procedure INIT and helps display messages. R\$RECEIVEMESSAGE is a system primitive that examines the mailbox to be serviced and places the token for the first

message there (MSG\$TOKEN). Another system primitive, R\$SENDMESSAGE, puts the token for the message into the terminal handler's output mailbox. The primitive R\$DELETESSEGMENT clears the used memory and returns it to the free-memory manager.

A postgraduate system

The lessons learned since 1977 about OEM requirements for executives have fueled the evolution of the RMX/80 system. This has led to the powerful new features of the RMX/86 system.

The RMX/80 began with a nucleus for the first single-board computer (ISBC 80/10). Subsequent boards contained more memory and offered higher throughput. Nuclei for the three later boards were made interchangeable, to act as the "software bus" for transporting applications software from one board to another. The RMX/86 solution is simpler: The new nucleus is hardware-independent so it can be used on future as well as current ISBC 86 boards.

So far, process-control designers have generally refused to add user-programming facilities, to assure that users do not interfere inadvertently with tasks handling critical process conditions. If a program dies during a chemical reaction, for example, a whole batch can be ruined and the processing facilities may have to be flushed out.

The RMX/86 system, however, incorporates facilities for keeping programs alive. Its nucleus contains an exceptional-conditions support. Actually, a powerful error-processing subsystem, it allows single-board computers to be made fault-tolerant with no adverse impact on throughput.

disks for later access by another system. However, the RMX/86 goes one step farther by setting up a hierarchical directory of files. This feature lets the user organize his files to be consistent with his application (Fig. 3).

The named-file system has another advantage: It permits file-access checking, so a user can decide which of his files he wants to protect and which to share with other users.

Each of the file options can be configured independently; the user may select the features he needs—neither more nor less. Furthermore, the user can add his own device drivers to the I/O system.

The RMX/86 is designed to offer the user a wide spectrum of convenient functions (Fig 4). For example, the user of mass-storage systems has a display directory and copy files available. Or, the OEM who needs his own interactive capability can easily extend the system's human interface routines to meet his requirements.

As μ P applications expand, so does the need for loaders that allow parts of the applications software to reside on disks. The RMX/86 package provides a resident system loader that permits loading for either absolute or relocatable format. ■

How useful?

Immediate design application
Within the next year
Not applicable

Circle No.

541
542
543

First, there's an option to specify to the nucleus whether or not there should be any error handling for a particular task. A programmer can write his own handler either to abort a task or to program a specific course of action (for example, report exceptional conditions and continue with next instruction; load copy of module and try again; start alarm program). The exceptional-conditions support also detects program errors such as a wrong call to the nucleus and system problems like insufficient memory. Naturally, the RMX/86 provides all normal OS functions. System options include a terminal handler for CRT and TTY console device drivers for Intel's floppy and hard-disk controllers and an integrated I/O system. A subsystem of the I/O system supports tree-structured hierarchical files (Fig. 3).

Most single-board computers are used with special digital devices and many with other kinds of files and media. So, the I/O system is designed to make it easier to add special files, new peripherals and custom device drivers—the user need never feel locked out. The RMX/86 I/O system provides the user with a very general file concept—as a data sink or source.

A Small-Scale Operating System Foundation for Microprocessor Applications

KEVIN C. KAHN

Abstract—Sound engineering methodology, which has long been valued in hardware design, has been slower to develop in software design. This paper uses a case study of a small real-time system to discuss software design philosophies, with particular emphasis on the abstract machine view of systems. It demonstrates how the currently popular software design axioms of generality and modularity can be used to produce a software system that meets severe space constraints while remaining relatively portable across a family of microcomputers. These sorts of constraints have often been used to justify *ad hoc* design approaches in the past. The results of the project suggest that the use of such techniques actually make the meeting of many constraints easier than would a less organized approach. In addition, the reliability and maintainability of the resultant product is likely to be better.

I. INTRODUCTION

A PROCESSOR, as defined only by its hardware, is typically not an adequate base upon which to build applications software. Broad classes of applications can be examined and found to share more than the hardware defined instruction set. To avoid the reengineering of this common functionality, we would prefer to build such common parts once and thereafter treat this base software as though it were part of the machine. For example, a software system sometimes called an operating system, an executive, a nucleus, a kernel, or some similar term, is often supplied with a hardware product and can be viewed in exactly this way. In this paper, we examine a small-scale system to demonstrate this approach to bridging the gap between the hardware and the application. That is, we will view the software as a direct extension of the hardware—a view which may indicate future directions in microprocessor integration of function.

This paper is meant as both a case study of a particular system design and as a suggestion of the proper approach to such design situations in general. We will first discuss the abstract machine view of computer systems and attempt to demonstrate that this is a useful philosophical approach for building systems. We will then apply this approach to the discussion of a system to coordinate programs performing real-time control functions—RMX-80TM [18]. The emphasis of the paper will be on techniques and methodology rather than on the particular functionality of RMX. Special attention will be given to such issues as the use of modularity to enhance the adaptability of the system and the use of design generality to achieve global rather than local optimizations.

II. THE CONCEPT OF ABSTRACT MACHINE

What is a computing "machine" or processing unit? We generally identify a processing unit as a particular collection of hardware.

Manuscript received September 1, 1977; revised October 11, 1977.

The author is with the Intel Corporation, Aloha, OR 97005.

TM Intel Corporation, Santa Clara, CA.

ware components that implement the instruction set of the machine. This very physical definition of a computer dates from mechanical processors. Even with modern computers, before large-scale integration, it was easy to physically point at the processing elements as distinct from memories, peripherals, and programs. Continued integration of function has at least made this physical distinction more difficult with single chips subsuming processing, memory, and peripheral interface functions. Microprogramming (i.e., replacing hardwired instruction logic with a more elementary programmed processor) as an implementation strategy has logically blurred this distinction as well. That is, when the basic visible instruction set of a processor is itself implemented in terms of more primitive instructions it is more difficult to identify "the machine." It is clear that this narrow physical definition of a processor is not adequate for current technology levels and is likely to become even less viable as the technology continues to develop.

Actually we have been using alternative definitions of a processor for some time. All of the theoretical work in finite state machines, for example, deals with conceptual processors. Likewise applications programmers seldom really regard the machine they program as much more than collection of instructions found in a reference manual—the physical implementation of the machine is of little concern to them. Indeed, they may never come physically near the hardware if they deal with a typical time-sharing system—rather, the terminal is the only physical manifestation of the computer such users may see.

More to point, perhaps, are the numerous interpreters that have been written for languages such as Basic. Each such interpreter actually produces a conceptual machine with one instruction set targeted to a specific application. With standard compiled languages such as Fortran, Algol, or Pascal, a higher level source statement is translated into the instruction set of the physical hardware. In contrast, interpreted language systems translate the source into the instruction set of some conceptual machine that is better suited to the running of programs written in the language. For example, the hardware may not provide floating-point instructions or define a floating-point data representation. In such a case it may be easier to define a machine that recognizes a particular floating-point data format with an instruction set that includes floating operations. These interpreters are high-level machines that have usually been implemented in software. Likewise, it should be readily apparent that, just as these interpreters provide high-level machines to their associated translators, any programming language, compiled or interpreted, provides one to its users.

Interpreters of this sort typically may examine and decode a stream of instruction values in a manner analogous to the hardware. Alternately, the new instructions may all be executed as subroutine calls using the appropriate hardware instruction. That is, the entire bit pattern for CALL *X* (where *X* is the address of a

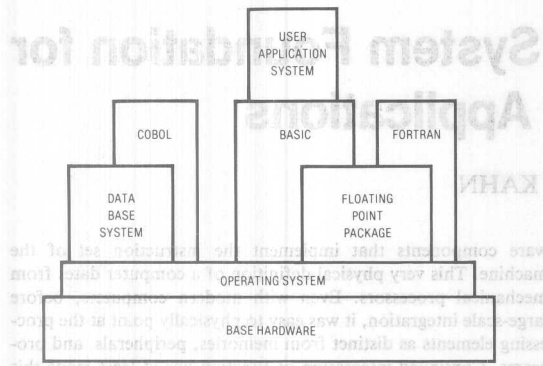


Fig. 1. Typical collection of abstract machines.

routine that implements a part of the new instruction set) can be regarded as a new operation code rather than as the hardware operation CALL. In either case the programmer using these extensions can view the hardware-software combination as though it were a new machine with a more useful instruction set. Microprogrammed machines such as the IBM 5100 or Burrough's 1700 have simply optimized the performance of such interpreters or subroutine packages by committing them to a faster storage medium.

Viewed in this light we can identify any collection of hardware and software that provide some well defined set of functions as defining an *abstract machine* [10],[12]. This machine has an instruction set that consists of the functions provided by the hardware-software combination. For a particular application it may be possible to view multiple such abstract machines by taking various pieces of the whole. For example, the physical machine provided by a set of components is just one abstract machine. It is of particular interest since it is the greatest common abstract machine that can be identified as being used by any application running on that computer system. A Basic interpreter running on this machine might then constitute a second virtual machine. A Basic program running on this interpreter that accepted high level commands and performed according to them might be a third level machine usable by people with no knowledge of either the hardware or Basic. Whenever we can identify functions of sufficient commonality among a number of applications, it may be worth viewing the software which provides these functions as extensions of the base hardware machine which define some augmented or even different machines. Users programming such an application can then view this abstract machine, rather than the base machine as the vehicle that they are programming, and in doing so avoid reengineering the functions that it provides. Fig. 1 illustrates an example of such machines. It is important to remember that at any time, many abstract machines may be thought of as existing on the same base hardware.

III. OPERATING SYSTEMS AS ABSTRACT MACHINES

The terms operating system or executive have been used to describe software systems of widely different functionality. These machines generally provide for the management of some machine resources such as input, output, memory space, memory access, or processor execution time. We might then attempt to define an operating system as some collection of software modules which defines an abstract machine that includes resource management functions as well as the hardware supplied computational func-

tions [2],[6],[8],[11]. With such a broad definition, however, large-scale multi-user time-sharing systems and small single user microprocessor development systems both may claim to have operating systems. Clearly, the range of software systems covered by this definition is large, encompassing products which differ by orders of magnitude in complexity. Rather than become involved in trying to resolve this disparity, we will qualify our use of the term and refer to an operating system "foundation." That is, we will describe a software system which provides a minimal base for the construction of real-time applications. We will avoid the somewhat irrelevant question of whether the system comprises a complete "operating system."

The important item to realize from the above discussion is that any operating system functionally enlarges the processor seen by the programmer. The functions that it provides become as much a part of the machine's functionality as jump instructions. Indeed, it is functionally unimportant to the user desiring to read from a file whether it requires a single hardware instruction or a large software routine to accomplish it. In terms of the abstract machine discussion above, we will examine a software package which defines an abstract machine that includes functions required to coordinate programs performing real-time control applications [1],[9],[12].

The key overall requirement of the operating system foundation that we discuss in this paper will be that it supply a minimal covering set of functions to permit coordination of asynchronous tasks. To determine this set we will need to further examine the needs of its users and environment of its use. In describing this foundation, we are defining an abstract machine that must be programmed to be of use; that is, like the instruction set of the base machine the foundation by itself performs no work but rather provides an environment within which useful tasks can be run.

We should note, here, some of the limitations of the system which differentiate it from large-scale operating systems. First, it is not primarily intended for a multi-user environment, particularly because the underlying hardware does not provide the necessary support to protect users from one another. Also, it will often be used to control functions of specialized devices and therefore is "close" to the I/O devices. That is, it does not supply the sort of high level I/O control system which is often present in larger systems for controlling more conventional I/O devices. Finally, it does not assume a backing store from which program overlays can be loaded (but it can easily support such an extension).

IV. DESIGN CONSIDERATIONS

A. Use Environment

The foundation system we will describe is RMX-80 [5] which was designed to be used with members of Intel's Single Board Computer (SBC) family of products. This family includes a wide range of bus compatible processor, memory, and peripheral boards. Of most interest to this discussion are the processor boards which are based on the Intel 8080 or 8085 microprocessors and include varying amounts of on-board ROM and RAM memory and I/O interfaces. In addition, the boards vary in the sophistication of their interrupt structures and timing facilities. In terms of abstract machines we might characterize these computers as essentially the same machine at the processor level but different machines at the computer system level. It was desired that the abstract machines defined by adding RMX to the underlying computers be as much the same as possible.

During the design of RMX, we expected that its users would span the entire broad range of applications across which the SBC

hardware was being put to use. This implied that it might see uses ranging from minimal single board systems that functioned as single device controllers to complex multiboard applications implementing involved real-time process or industrial control functions. In particular we expected that many user-built I/O boards and peripherals would be used with the system. It was important for us to allow full use of these unknown devices with RMX while still providing as much assistance as possible in the building of the controlling software systems.

As is the case with most processors, the concrete (i.e., physical) machines represented by the SBC family do not themselves include any facilities to permit multiple asynchronous functions to be programmed, to provide for the coordination of such functions, or to provide time information needed for real-time applications. Typically, users of these products have directly programmed these functions in an *ad hoc* manner within their applications. An examination of the sorts of functions necessary to such applications reveals that at the very least this reengineering is a waste of resources. Worse is the high probability of error in programming such critical functions.

The SBC hardware products were designed to eliminate the complexities of board engineering, particularly for those users without the necessary expertise to handle the task, by functionally integrating individual components into complete boards. The programming of functions to coordinate parallel software activities is, likewise, an area which should be carefully engineered in order to avoid subtle errors. The development of RMX was therefore viewed as a process of functional integration analogous to the integration of LSI components into boards. That is, just as a well designed board relieves the user of component level hardware engineering, RMX relieves the users of low-level software engineering.

B. System Requirements

The hardware environments and anticipated uses of RMX defined a stringent set of requirements for it. Foremost among these were its memory constraints; indeed, for the anticipated uses, memory size considerations dominated execution speed ones over a considerable range. Since we expected applications that would reside entirely on a single board with 4K bytes of PROM, the maximum size of the RMX foundation code was set at half of this or 2K bytes. Further, unlike larger minicomputer systems, many, if not most, applications of the SBC boards would not have available any mass storage or other program loading device. It was thus important that RMX be designed to be ROM (or PROM) resident and capable of automatically initializing the system when powered on.

We also anticipated that the expertise of many RMX users would be in areas other than programming systems. We therefore felt that the RMX machine needed to provide a fairly simple set of concepts, avoiding where possible those constructs most likely to cause errors. For example, we felt that a very frequent source of programming difficulty lay in dealing with interrupts. Many latent errors in programming systems stem from the occurrence of an interrupt at an unexpected time. We therefore decided to attempt to minimize the need for users to deal with hardware interrupts or with the interrupt-like occurrences found in many minicomputer operating systems. At the same time we had to accommodate the needs of the sophisticated user who still desired to take advantage of RMX but had a specific need to directly control the hardware via the interrupt facility.

Finally, to define the general functionality of RMX we examined its anticipated applications. Real-time applications commonly need to perform a number of tasks of differing importance

logically in parallel, with preference always being given to executing the most critical ones first. While these tasks may be relatively independent, they may need to periodically synchronize themselves with one or another distinct task or with the outside world. For the latter, interrupts are the usual hardware supplied mechanism. Some tasks may also need to communicate data with one another. For example, a task servicing a sensing device may take readings from the device which need to be communicated to two tasks: one task which reacts to the reading by controlling some other device, and another task which logs or tabulates the readings. Ranked in order of importance these might be control, sensing, and logging. Finally, the tasks must have the ability to control themselves relative to real-time, either by delaying their execution for certain periods or by guaranteeing that they are not indefinitely delayed by, for example, a faulty device.

Requirements on the system design were also generated by considerations internal to the design project. One of these was the need to provide a single RMX abstract machine on a variety of underlying SBC boards. While separate versions of RMX for each board could have been designed with the same external appearance, this approach would have led to an unnecessary amount of internal engineering. Additionally, without careful initial design, the differences in the base hardware would have had visible effects on the RMX abstract machine for each of the boards. This requirement demanded that we partition the structure of RMX into two parts. One part would implement those aspects which were independent of the particular hardware. The second part would interface the first part to the underlying hardware of the specific boards [7].

We also wished to minimize the software development costs by applying the best available software engineering techniques. Historically, tight space constraints have often led to a very *ad hoc* approach to software design in the belief that more generally designed external features or more modularly built internal designs would lead to inherently larger systems. As a result of this philosophy, each needed function is designed to be as small as possible. Unfortunately, while each function may be locally optimized, it is possible that the overall design suffers from duplication or overlap between such individual elements. Current work in programming methodology stresses modularity, generality, and structure (most often for their side effects in producing more maintainable, less error prone systems).

We felt that there was more to gain, both in development cost and space performance, by avoiding optimized specialization of function in favor of more general designs [17]. This reduced the number of separate functions that RMX had to supply. The resulting external design therefore has a single mechanism that provides task communication, synchronization, time references, and standard interrupt-like control. To do so it incorporates the operating system design approaches favored in much of the modern computing literature. Likewise, the internal structures are highly modular and designed to be as uniform as possible so as to avoid replicating similar, but nonidentical internal management routines.

V. THE RMX MACHINE

B. General Concepts

The abstract machine defined by RMX augments the base microprocessor by introducing some additional computational concepts. We define a *task* to be an independently executable program segment. That is, a task embodies the concept of a program in execution on the processor. RMX permits multiple tasks to be defined which can run in a parallel, or multiprogrammed,

fashion. That is, RMX makes individual tasks running on one processor appear to be running on separate processors by managing the dispatching of the processor to particular tasks. The registers on the processor reflect the activity or state of the running task. Other tasks may be ready to execute but for some reason have not been selected to run yet and so have their processor states saved elsewhere in the system. From the point of view of the program that is a task, execution proceeds as though it were the only one being run by the processor. Only the apparent speed of execution is affected by the multiprogramming. From the point of view of the system, every task is always in one of three states: running, ready, or waiting. The task actually in execution is running. Any other task which could be running but for the fact that the system has selected some other task to actually use the processor, is ready. Tasks which are delayed or stopped for some reason are waiting, as will be discussed below.

Each task is assigned a *priority* which determines its relative importance within the system. Whenever a decision must be made as to which task of those that are ready should be run next, the one with the highest priority is given preference. Furthermore, in the spirit of unifying mechanisms, the same priority scheme replaces a separate mechanism for disabling interrupts. Interrupts from external devices are logically given software priorities. If the applications system designer deems a particular task as of more importance than responding to certain interrupts, he can specify this by simply setting the RMX priority of that task to be higher than the RMX priority associated with those given hardware interrupts. It is thus possible to maintain a high degree of control over the responsiveness required for various functions.

As mentioned above, tasks may desire to communicate information to one another. To this end the RMX machine defines a *message* to be some arbitrary data to be sent between tasks. To mediate the communication of messages it defines an *exchange* to be the conceptual link between tasks. An exchange functions somewhat like a mailbox in that messages are deposited there by one task and collected by another. Its function is complicated by the fact that a task may attempt to collect a message at an exchange that is empty. In such a case the execution of that task must be delayed until a message arrives. Tasks that are so delayed are in the waiting state. We chose this indirect communication mechanism over one which directly addresses tasks because it permits greater flexibility in the arrangement of receiver and sender tasks. The anonymity of the receiving task implies that the sender need know only the interface specification for a function to be performed via a message to a particular exchange. The task or tasks which implement that function need not be known and hence may be conveniently changed if desired.

The conventional mechanism used by programs to communicate with external devices is the interrupt. Unfortunately, interrupts are by nature unexpected events and programming with them tends to be error prone. The essential characteristic of an interrupt is that a parallel, asynchronous activity (the device) wishes to communicate with another activity (a program). Since this communication is essentially the same as that desired between separate software tasks it seems conceptually simpler to use the same message and exchange mechanism for it. The unification of all communications functions is analogous to the idea of standardized I/O found in systems such as UNIX [17]. The RMX machine eliminates interrupts by translating them into messages which indicate that an interrupt has occurred. These messages are sent to specific exchanges associated with particular interrupts. Tasks which "service interrupts" do so in RMX by attempting to receive a message at the appropriate exchange. Thus, prioritized nested interrupts are easily handled. An advantage of this unified

treatment of internal and external communication is that hardware interrupts can be completely simulated via another software task. This facilitates debugging and permits easy modification of a system by allowing rather arbitrary insertion of tasks into a network of communicating tasks and devices.

Note that with this scheme unexpected interrupts do not cause particular difficulty. For example, if the servicing task is still busy with some previous message, the interrupt message will be left at the exchange and will not affect the task until it is ready for another interrupt; i.e., until it waits at the exchange. In an application designed to properly handle the actual interrupt rate, the task will service interrupts quickly enough to always be waiting when the next one occurs. In this case, response to an interrupt is immediate. Thus this mechanism provides no loss of facility relative to the usual interrupt scheme but it does make the proper controlling of such events simpler. Multiple occurrences of the same interrupt which indicate the processor has fallen behind in its servicing are logged as such by a message which indicates that interrupts may have been lost. These interrupts do not, however, disrupt the running task or complicate programming.

The last concept embodied in the RMX abstract machine is that of time. The RMX machine defines time in terms of *system time units*. It then permits tasks to delay themselves for given periods of time so that they can synchronize themselves with the outside world. It also permits tasks to guard against unduly long delays caused by attempting to collect a message at an empty exchange by limiting the length of time that they are willing to spend waiting for some message to arrive.

B. Data Objects and Functions

These concepts are realized in RMX by introducing some new data objects and instructions. Just as the base processor can deal directly with such data objects as 8 bit bytes or unsigned integers, the RMX abstract machine can deal directly with the more complex data objects: task, message, and exchange. Each of these data objects consists of a series of bytes with a well defined structure and may be operated upon only by certain instructions. This is completely analogous, for example, to a machine that permits direct operations on floating-point data objects which consist of four bytes with a particular internal structure to represent the fraction, exponent, and signs. In each case there are only certain instructions that can be used correctly with the object and the internal structure of the object is not of particular interest to the programmer.

The new instructions provided by RMX are: SEND, WAIT, ACCEPT, CREATE TASK, DELETE TASK, CREATE EXCHANGE, and DELETE EXCHANGE. The create instructions accept blocks of free memory and some creation information to format and initialize the blocks with the appropriate structure. Each corresponding delete instruction accepts one of the objects and logically removes it from the system. The remaining operations are of more direct interest to the operation of the RMX machine.

The WAIT instruction has two operands: the address of an exchange from which a message is to be collected and the maximum time (in system units) for which the task is to await the arrival of a message. The result of the operation is the address of the message which was received. A special message from the system indicates that the specified amount of time elapsed without the arrival of a normal message. From the programmer's point of view this instruction simply executes and returns the specified result. Actual execution of the instruction will involve the delaying of task execution if no message is available, by queueing it in a first-come-first-served manner at the exchange. Any such delay is not visible

to the programmer, however. This approach unifies the communication and timing aspects of the design. It directly provides reliability in the face of lost events due to hardware or software failure. Tasks can be guaranteed not to be indeterminately delayed due to such failures and can thus attempt recovery from them. It also permits tasks to use the same mechanism to delay themselves for given time intervals by waiting at an exchange at which no message will ever arrive.

The ACCEPT instruction is an alternate way to receive a message. It has a single operand specifying the exchange from which the message is to be received and immediately returns either the next message at the exchange or a flag indicating that no message was available. The task is never delayed to await a message in the ACCEPT operation.

SEND also has two operands: the address of a message and the address of an exchange to which the message is to be sent. The instruction queues the message in a first-come-first-served manner at the exchange if there is no task already waiting there. If a task is waiting at the exchange then the instruction binds the message to the task and makes the task eligible to execute on the processor. When the receiving task resumes actual execution the address of the message will be returned to it as the result of its WAIT instruction.

VI. THE RMX IMPLEMENTATION

A. Methodology

In this section and the next, we consider some (but certainly not all) details of the actual implementation of the system as illustrations of the design of such software products. We turn first to the methodology applied to the effort and then to some samples of the mechanisms.

To provide the abstract machine just described and meet the other requirements for the system, RMX was implemented as a combination of ROM resident code and some RAM resident tables. Just as a hardware designer uses LSI devices in preference to more elementary TTL components, we chose to use the leverage of a high level programming language rather than elementary assembly code. The system was, therefore, designed using PLM [14], Intel's high-level implementation language. The operations described above appear as procedure calls using the standard PLM calling sequence. The space constraints and a good level of internal maintainability were achieved by maximizing the modularity of the design. The broad independent functions of multiprogramming, communications and control were completely isolated from the board dependent timing and interrupt handling functions. As a result, movement of the system to a new member of the SBC family requires only the reimplementing of these board dependent functions. In addition, data structure of internal and user visible objects were generalized so that single algorithms could deal with any of them. Individual optimizations could have been made in the local design of many parts of the data structures to improve their space or time costs slightly. Such optimizations, however, would have cost considerably more in code space and code complexity [3].

The module feature of PLM was used to simulate the abstract data type concept [4],[13] and enforce information hiding [15],[16]. That is, every data structure used by RMX is under the exclusive control of a single module. The modules supply to each other restricted sets of public procedures and variables. It is only through these paths that agents outside a module may access the data structures maintained by the module. The only assumptions that such outside agents may make about a module and its data structures are those specified by the definition of the public paths.

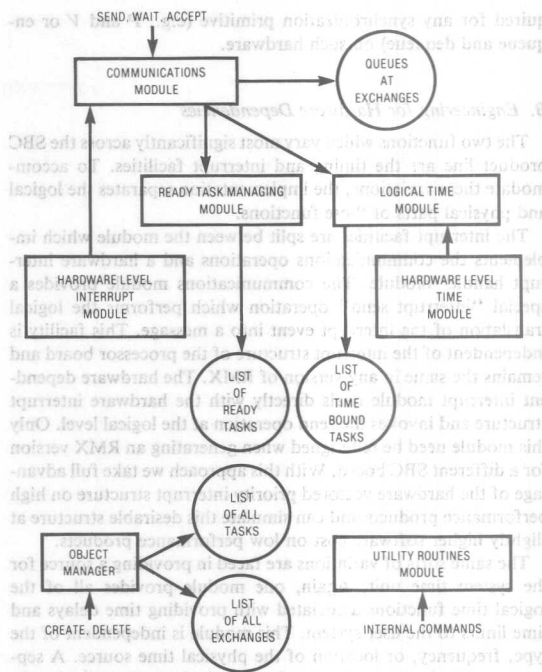


Fig. 2. Major modules (boxes) and data structures (circles) of RMX.

As a result, so long as these interface specifications are maintained, any given data structure may be reorganized by redesigning its controlling module without affecting other parts of the system. This approach improves the understandability of the implementation and facilitates the debugging and maintenance of the system. Fig. 2 illustrates the general structure of the RMX implementation.

Finally, the original version of RMX was completely coded in PLM using the resident PLM compiler of the Intel® Microcomputer Development System. This version was functionally complete but slightly exceeded the space constraints, occupying about 2.5K bytes of program space. There were a couple of cases where the language structure of PLM did not permit the direct expression of the best way to compile the code. For these modules, it was sufficient to hand optimize the code output by the compiler. The original structure of the PLM program was maintained and the majority of its generated code was used intact. The final RMX system occupies less than 2K bytes of program space. This high level language approach coupled with selective manual optimization permitted far quicker and more error free development than could have been achieved using assembly language.

The approach to handling interrupts did introduce additional software overhead. For a typical configuration of the hardware, the realistic minimum interrupt latency would be about 200 μ s. Using the message mechanism it is about 800 μ s. For the targetted process control applications, this is entirely acceptable. RMX does make provision, however, for direct handling of selective interrupts which require better response time without disturbing the use of the message mechanism for the others. For normal task communication, the performance is relatively better. For the typical hardware configuration, the transmission of a message takes about 800 μ s, which is comparable to the time that would be re-

quired for any synchronization primitive (e.g., *P* and *V* or enqueue and dequeue) on such hardware.

B. Engineering for Hardware Dependencies

The two functions which vary most significantly across the SBC product line are the timing and interrupt facilities. To accommodate these variations, the implementation separates the logical and physical parts of these functions.

The interrupt facilities are split between the module which implements the communications operations and a hardware interrupt handler module. The communications module provides a special "interrupt send" operation which performs the logical translation of the interrupt event into a message. This facility is independent of the interrupt structure of the processor board and remains the same in any version of RMX. The hardware dependent interrupt module deals directly with the hardware interrupt structure and invokes the send operation at the logical level. Only this module need be redesigned when generating an RMX version for a different SBC board. With this approach we take full advantage of the hardware vectored priority interrupt structure on high performance products and can simulate this desirable structure at slightly higher software cost on low performance products.

The same sorts of variations are faced in providing a source for the system time unit. Again, one module provides all of the logical time functions associated with providing time delays and time limits to the user system. This module is independent of the type, frequency, or location of the physical time source. A separate module is responsible for clocking the logical level by invoking it once every system time unit. Once again, this permits a consistent definition of time in RMX systems regardless of the sophistication of the available time source, and it limits the amount of reimplementations that is needed to support new SBC products.

C. Example Data Objects

As an example of the complex data objects defined in the system we will consider the task and exchange objects illustrated in Fig. 3. The task object is 20 bytes long and embodies the execution state and status of a task. It consists of pointers used to link it onto various lists of tasks in the system. These lists are used to queue a task at an exchange, link it to other ready tasks, or keep track of its maximum delay when waiting. It also contains the stack pointer of non-running tasks which is sufficient to supply the remaining task register values when the task next executes. Finally, the object contains the task priority, some status information describing the state of the task, and a pointer to auxiliary information about the task.

The exchange object is 10 bytes long and implements the mailbox concept described earlier, primarily by serving as the source of header information for lists of messages and tasks. Each of these singly linked lists is addressed with head and tail pointers located in the exchange object. All exchanges in the system are also linked together.

The exchange objects are operated upon by the SEND, WAIT, and ACCEPT instructions of the RMX abstract machine. These instructions generally alter the "value" or contents of these complex data objects. The task object is not the direct operand of any RMX instruction described above. Rather it is indirectly altered as a side effect of various instructions. Just as the user of floating-point objects on most machines needs to know the length and existence of instances of the object, but not its internal structure, so the internal structure of these objects is generally unimportant to the users.

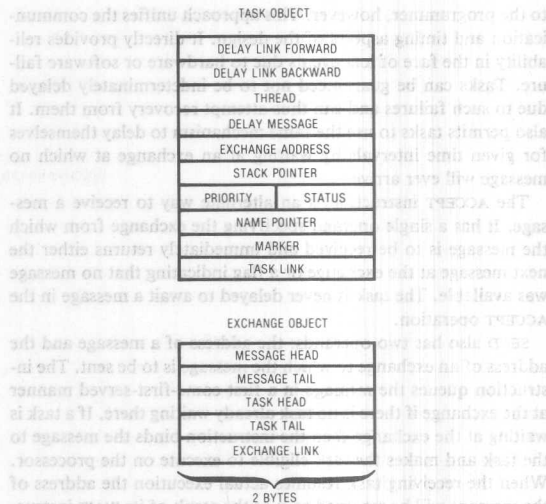


Fig. 3. Example data objects in RMX.

D. Global Versus Local Optimizations

We have already discussed some aspects of global versus local optimizations at the overall design level in terms of avoidance of redundant features. A good example of this tradeoff in the implementation is provided by the linked list data structures within RMX. Like many such systems there are a number of singly linked lists which must be maintained to reflect the status of the system. Local optimizations on the placement of links within data structures or in the form of the headers used for the lists would be guaranteed to save a few bytes of data space across the various lists. Further, the list insertion, scanning, and deletion algorithms could be specially tailored to the individual list structures to save microseconds of execution time for some operations on some lists. Indeed, any one such tailored algorithm might well use less code space than a single more general one.

On the other hand, many of the list operations are in no sense time critical. Generalizing all the list structures to use a single form replaces multiple algorithms with one, thus saving code space. The particular form can be chosen to favor those operations that are frequent, thus limiting the impact of the generalization on the execution speed of the system. Perhaps most important, however, is that, by reducing the number of algorithms and structures used, we decrease the potential number of errors and improve the maintainability of the resultant product. Since there are, for example, at least six distinct singly linked structures in the system, we reduce overall code size and engineering cost by supporting only a single mechanism. We improve product reliability at the price of a small increase in fixed data space and a small execution speed penalty of infrequent and nontime-critical operations.

It is interesting to note as an aside that this is really an example of software engineering: that is, applying engineering discipline to software development. Such discipline is highly valued and understood in other engineering fields. Standardized mechanical or electrical components are virtually always preferred to special designs; PLA's often replace random logic. Unfortunately, an appreciation of the overall benefits of such structure has been slow to develop in software engineering. Too often, we have seen special purpose designs and overly complex structure used in pro-

grams supposedly to save space or improve speed. The true costs in development time and reliability of such approaches have often been underestimated; the true time savings attributed to them often overestimated. The high percentage of end product cost due to software is finally forcing a general awareness of these issues.

VII. LSI AND ABSTRACT MACHINES

It seems natural at this point to ask how the abstract machine view of systems in general and our experience with RMX might be affected by the continuing development of LSI technology. Once we view any complex software system as defining a collection of abstract machines, it becomes obvious that it is simply an engineering decision as to which machines should be committed to hardware. We are constrained in this choice by the densities of our solid-state technology, the performance we desire, the applications that we are attacking, and perhaps most severely, by our understanding of software systems and of the machine structures that they require.

We might build an entire final application (e.g., a cash register) as a very-high-level single-chip machine. The specialization of such a design would, however, severely limit its application beyond the one for which it was specifically meant. On the other hand, we could build exclusively bit slice microprogrammable machines with utmost generality but which, due to their very low level of functional integration, would have no technological leverage for attacking complex problems. Actually, both these extremes have their well developed roles and will continue to be reasonable approaches for high-volume low-cost, and special-purpose tailored systems, respectively. It is in the middle ground—the area of the traditional computer—that directions are less clear.

If the 8080 type processors are generally somewhat less powerful than we actually need and as a result we always build operating systems of some level to support them, perhaps some of these functions can be integrated into the hardware. That is, if we can identify a broad range of systems which include essentially the same abstract machine implemented in software, then that abstract machine is a good candidate for hardware integration. The engineering difficulty is in understanding these software structures well enough to confidently and correctly commit them to hardware.

Attempting to build all of some very large and complex operating system onto one or two chips is, no doubt, out of the question with current technology. On the other hand, the final RMX system which we described resides in a small amount of ROM within the 65K address space of the 8080 processor. Once we view RMX as an abstract machine, the placement of the code which implements its functionality becomes immaterial. In particular, we could build an augmented 8080 type processor directly by defining the additional instruction codes of RMX as hardware operations and moving the RMX implementation into microcode on the chip. The resultant component would indeed be an "RMX machine" which dealt directly with the complex data objects and tables described above. It would have the advantage of not using any of the address space for operating system code. More importantly, it would not waste bus cycles and memory access time fetching operating system instructions. Such a machine would have the same advantages over a conventional one that a machine with floating-point hardware has over one without it.

Should we then try to build the RMX machine—ignoring for the moment whether our hardware technology is capable of it quite yet? Is the simple task model of RMX sufficiently general to be of use over a wide class of applications? Is the RMX machine the complete tool that we would like? Clearly, the answer is not a

wholehearted yes. For one example, RMX provides no isolation or protection of one task from another. Indeed, no solely software system can provide such protection at any reasonable cost. Such isolation would be desirable at the least because it would limit the damage that one task could do to another due to errors. The conclusion to be drawn, therefore, is not that this particular abstract machine should be built in hardware, but rather that some such machine would provide more of the facilities needed for building microprocessor applications than do current processors. Further, the design principles discussed above are the ones that appear most likely to be fruitful in creating such a machine.

VIII. CONCLUSIONS

In this paper, we have attempted to use a case study of a particular small operating system to illustrate both a philosophical approach to viewing computer systems and some important aspects of software development methodology. Many of the subtle aspects of designing software to control quasi-parallel activities have not been discussed in detail, nor have we fully described the implementation. Nevertheless, we hope that this description suggests the practicality and necessity of disciplined approaches to software system design. Until software implementation reaches a level of engineering commensurate with that applied to other aspects of computer system design, our products will be very much bound by software costs. Only discipline and structure within our software efforts will ultimately permit microprocessor applications to reach their full potential.

ACKNOWLEDGMENT

The author acknowledges the effort of codesigner K. Burgett in the original development of the system. In addition, thanks are due for the detailed suggestions received from J. Rattner, S. Fuller, R. Swanson, G. Cox, and J. Crawford, which greatly improved the content and clarity of the paper. The author also thanks his other colleagues at Intel and the reviewers who contributed to the final form of the paper.

REFERENCES

- [1] P. Brinch Hansen, "The nucleus of a multiprogramming system," *Commun. ACM*, vol. 13, no. 4, pp. 238-241, Apr. 1970.
- [2] —, *Operating System Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
- [3] F. P. Brooks, Jr., *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.
- [4] W. L. Brown, "Modular programming in PL/M," in *Proc. IEEE Conf. Computer Software and Applications*, Nov. 1977.
- [5] K. Burgett and E. F. O'Neill, "An integral real-time executive for microprocessors," *Computer Design*, vol. 16, no. 7, pp. 77-82, July 1977.
- [6] E. G. Coffman, Jr., and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
- [7] G. W. Cox, "Portability and adaptability in operating system design," Ph.D. dissertation, Purdue Univ., Lafayette, IN, Dec. 1975.
- [8] P. J. Denning, "Third generation computer systems," *Computing Surveys*, vol. 3, no. 4, pp. 175-216, Dec. 1971.
- [9] E. W. Dijkstra, "The structure of the 'THE'-multiprogramming system," *Commun. ACM*, vol. 11, no. 5, pp. 341-346, May 1968.
- [10] J. H. Fasel, "Abstract machine hierarchies for programming language implementation," Ph.D. dissertation, Purdue Univ., Lafayette, IN, Dec. 1977.
- [11] A. N. Habermann, *Introduction to Operating System Design*. Chicago, IL: SRA, 1976.
- [12] A. N. Habermann, L. Flon, and L. Coopridge, "Modularization and hierarchy in a family of operating systems," *Commun. ACM*, vol. 19, no. 5, pp. 266-272, May 1976.
- [13] B. Liskov and S. Zilles, "Programming with abstract data types," *SIGPLAN Notices*, vol. 9, no. 4, pp. 50-59, Apr. 1974.
- [14] D. D. McCracken, *A Guide to PL/M Programming for Microcomputer Applications*. New York: Wiley, 1977.
- [15] D. Parnas, "A technique for software module specification," *Commun. ACM*, vol. 15, no. 5, pp. 330-336, May 1972.
- [16] —, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053-1058, Dec. 1972.
- [17] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Commun. ACM*, vol. 17, no. 7, pp. 365-375, July 1974.
- [18] *RMX/80 System Users Guide*. Santa Clara, CA: Intel Corp., 1977.

May 1981

Executive Pros

...a 16-bit data bus with IAPX 86 and IAPX 286. It also supports Intel's iSBX 286 and iSBX 386 single-board computers. The 286 is fully compatible—not only procedure-by-procedure but also for all hardware addresses and interrupt vectors. It also can be used with custom boards that contain an IAPX 86 or IAPX 286. The good news is that the programmable interrupt controller and 8259 programmable interrupt timer are 100% and 80% programmable, respectively. In addition, the 8253 programmable-counter-timer is 100% programmable and the most important—an 8087 coprocessor is optional.

Jess M. Irwin
Electronic Design

"Reprinted with permission from *Electronic Design*, Vol. 29, No. 5, copyright Hayden Publishing Co., Inc. 1981"

Faster and smaller than its predecessor, the iRMX 88 real-time modular operating system eases the transition from 8-bit to 16-bit μ Cs, while supporting the 8087 arithmetic chip and other peripherals.

Multitasking executive speeds 16-bit micros

Even while it eases the transition from an 8-bit to a 16-bit world, a new operating system (OS) brings thoroughbred performance to the stable of modular systems software. The iRMX 88 multitasking executive cuts both interrupt latency (the delay until a random event gets a response) and context-switching time (the interval during which the processor changes from one task to another). Its nucleus is, therefore, faster—and smaller—than those of preceding operating systems.

The iRMX 88 Executive provides fundamental multimasking software and complements the full-featured, multiprogramming iRMX 86 OS (ELECTRONIC DESIGN, March 15, 1980, p. 245). Through priority-based task scheduling, it concurrently monitors and controls multiple events. It provides real-time clock control, manages interrupts, and dispatches tasks. As an upgraded version of the 8-bit iRMX 80 Executive, iRMX 88 employs the same concepts and most of the same modules. Thus, the new OS offers field-proven and reliable interfaces.

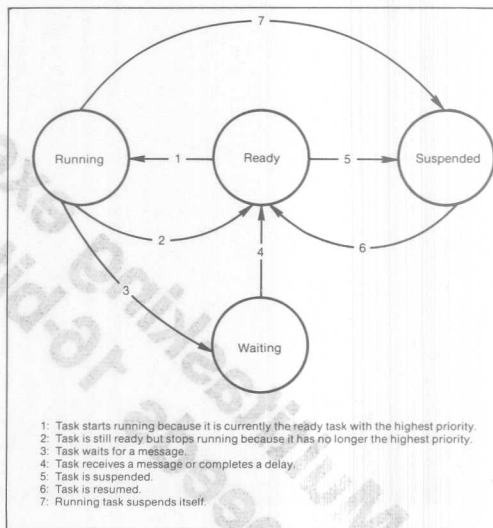
Tasks share resources

The Executive will run with iAPX 88 or iAPX 86-based boards and supports Intel's iSBC 86/12A, iSBC 88/40, and iSBC 86/05 single-board computers. Since it is fully configurable—not only procedure-by-procedure, but also for all hardware addresses and interrupt levels—it also can be used with custom-designed boards that contain an iAPX 86 or iAPX 88 processor, an 8259A programmable-interrupt controller, and 8253 programmable-interval timer, an optional 8251A programmable-communications interface, and—perhaps most important—an 8087 numeric-data processor.

In a multitasking system like the iRMX 88, several

tasks execute concurrently. A task is an independent program that shares system resources and communicates with other tasks. Each task has its own stack, where the registers used by the task are saved to shorten context-switching time.

A task communicates with other tasks via messages, which convey data and synchronization information. These messages are channeled through data structures called exchanges. A task can send messages to an exchange (using the RQSEND procedure) or wait for messages at an exchange (using the RQWAIT procedure). In the latter case, the task does not execute until a message is available or until a certain length of time has elapsed, specified in terms of "system time units" (minimum interval of real time recognized by the system). A task that is



1. Tasks in the iRMX 88 Executive occupy one of four states. Transitions are governed by the seven occurrences listed at the bottom of the figure.

Jess M. Irwin, Project Leader
 Intel Corp.
 5200 N.E. Elam Young Pkwy, Hillsboro, OR 97213

IRMX 88 Executive

not willing to wait may accept any message from an exchange (using the RQACPT procedure) if one is available; otherwise, the task remains without a message.

Each task exists in one of four states: running, ready, waiting, and suspended (Fig. 1). A task is running if the processor is executing instructions on its behalf. It is considered ready if it is either running or able to run. A task is waiting if it has requested a message but has not received it. Finally, a task is suspended if some other task has specifically requested that it not be permitted to run.

Each task has a permanent priority, which indicates its importance relative to other tasks in the system and to interrupts from peripherals. Priorities range from 0 to 255, with 0 being the highest priority. The "idle task" executes at priority 255 and prevents any other task at that priority from running. The ready task with the highest priority is the running task. The software priorities correspond to eight hardware interrupts (Fig. 2). When the running task has a high priority, interrupt levels associated with equal or lesser priorities are masked off.

Interrupts provide real-time access

An interrupt invokes an interrupt-service routine, which either requests an end of interrupt (using the

RQENDI procedure) or requests that a message be sent (using the RQISND procedure) to an interrupt exchange associated with the active interrupt's level (Fig. 2). In the latter case, an interrupt task waiting at the interrupt exchange would complete the interrupt servicing. The system can enable interrupts (via the RQELVL procedure) and disable them (via the RQDLVL procedure) and can set the interrupt vector associated with a given priority level (using the RQSETV procedure).

The iRMX 88 treats interrupts as messages. When an interrupt occurs, the Executive creates a special message and sends it to the exchange associated with that particular interrupt. Should a previous interrupt message be at the exchange when the interrupt occurs, the message is changed to indicate a missed interrupt. Any task awaiting the interrupt exchange receives the interrupt message and is readied for execution. If the task has a priority corresponding to the interrupt level, it will become the running task. The task that was running when the interrupt occurred is still ready to run, but is pre-empted. When the running task relinquishes control by waiting at an exchange (possibly for another interrupt), suspending itself (via the RQSUSP procedure), or deleting itself (via the RQDTSK procedure), the iRMX 88 dispatches the next ready task.

Interrupt-processing tasks are simply tasks that wait at interrupt exchanges. They do not need to save the interrupted task's state or control the interrupt hardware. A task may simulate an interrupt by sending a message to an interrupt exchange (using the RQSEND procedure).

Because the iRMX 88 imposes some software overhead, a direct interrupt-servicing facility is provided for those cases where interrupt response time is critical. The interrupt vector is set with a pointer to an interrupt-service routine (using the RQSETV procedure); then all interrupts occurring at that level are handled by the interrupt-service routine, which is also responsible for all interrupt-logic control and state-saving. To complete the interrupt, the routine must use the RQISND or RQENDI procedure.

Creating tasks and exchanges

The iRMX 88 supports the dynamic creation of tasks and exchanges, as well as their deletion when they are no longer needed. Creating a task (with the RQCTSK procedure) places the new task on the system's list of ready tasks, according to its priority. The creation of an exchange (using the RQCXCH procedure) results in the initialization of an "empty" exchange—one that has no waiting messages or tasks. The deletion of a task (via the RQDTSK procedure) results in its removal from all system lists. The deletion of an exchange (using the RQDXCH

Hardware	Software
	Level 0 Highest priority
Level 0	1
	16
	17
1	32
	33
2	48
	49
3	64
	65
4	80
	81
5	96
	97
6	112
	113
7	128
	129
	255 Lowest priority

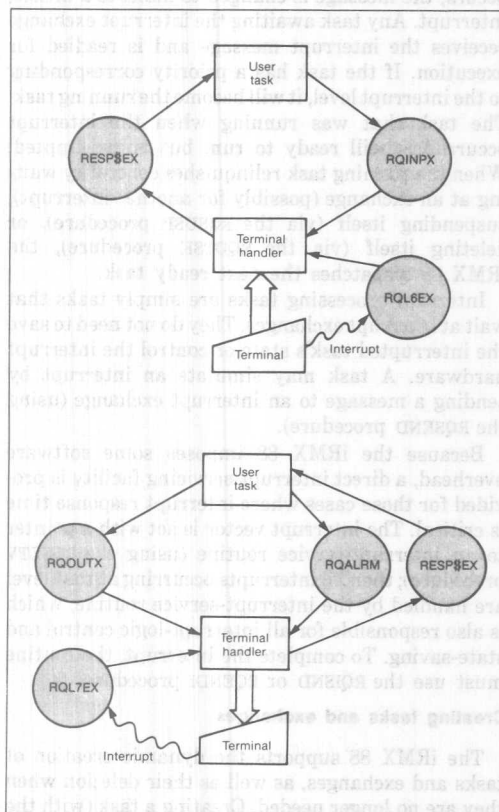
2. Of 256 interrupt levels, the top 129 are associated with hardware. The remainder can be assigned to software tasks.

procedure) is only possible if no messages or tasks are waiting at the exchange.

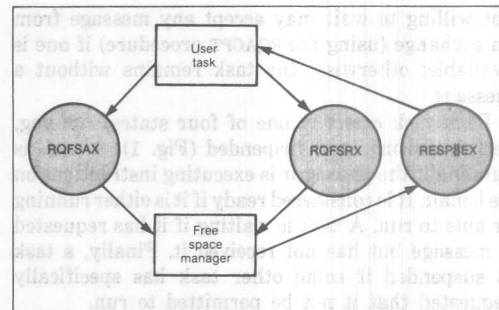
Suspension of a task (using the RQSUSP procedure) places the task on a list of suspended tasks and removes it from the list of ready tasks; to return the task to the ready state, the RQRESM procedure must be used.

A coprocessor for number crunching

Unlike previous operating systems, the iRMX-88 can support the 8087 numeric-data processor (NDP) in a way that is invisible to the user. When an NDP task is created, the iRMX 88 initializes and maintains the state of the 8087 in a save area. Should an



3. Interaction between the terminal and a user task involves several iRMX 88 exchanges (gray). Some serve for inputs from the keyboard (top); others, for outputs to the CRT (bottom).



4. When a user task needs a block of RAM, exchange RQFSAX passes the request to the free-space manager, which sends an acknowledgment via RESP#EX. The RQFSRX exchange serves to release the block.

DESTINATION:	POSITION:
1.0000 METERS	X = 1.2000 METERS
4.0000 METERS	Y = 3.4000 METERS
5.0000 METERS	Z = 2.1000 METERS
MAXIMUM VELOCITY:	VELOCITY:
0.0100 METERS/SECOND	DX = -0.0050 METERS/SECOND
0.1000 METERS/SECOND	DY = 0.0800 METERS/SECOND
0.0500 METERS/SECOND	DZ = 0.0500 METERS/SECOND

5. The arm of an industrial robot can move in three directions: X, Y, and Z. The console display reports current position and velocity (left), as well as the limiting values (right).

unmasked error occur in the 8087, an interrupt is generated on a configurable level. Using the default interrupt-service routine, the iRMX 88 sends an interrupt message to the interrupt exchange; the user can then supply his own error handler. The error task must specify itself as a non-NDP task or the system will deadlock. But the error interrupt, if used, must not be masked off.

The Executive includes a terminal handler which establishes real-time asynchronous serial communication between an operator's terminal and the tasks running under the executive (Fig. 3). The terminal handler provides a simple operator interface, which includes line-editing features, a 64-byte type-ahead buffer, and control over the output.

Since all port addresses and interrupt levels for the terminal handler are configurable, it supports any component configuration using an 8251A programmable-communications interface with an 8253 programmable-interval timer for baud-rate generation. A task interfaces with the terminal handler through two exchanges, where messages are sent to initiate read and write requests; they are processed on a FIFO basis. Characters are read and written in response to interrupts from the 8251 programmable-communications interface. Each request results in the transmission of one line to or from the terminal. Read requests are sent to the RQINPX exchange, and write requests to the RQOUTX

IRMX 88 Executive

exchange. When a request has been serviced, its message is sent to the specified response exchange.

The iRMX 88 also contains a free-space manager, which maintains a pool of free RAM memory (Fig. 4). Tasks request blocks of memory from the pool and return blocks to the pool. The request is made by sending a message to the manager's allocation exchange, RQFSAX, specifying the length of the needed block. If sufficient contiguous free RAM is available, the requesting task receives a message acknowledging the request and supplying the address of the block in the form of a message. Otherwise, the requesting task receives a message indicating how much free RAM can be allocated. The requesting task may then ask for a smaller block. In either case, the response from the free-space manager is returned in the same message that was used for the request. If the task must have the block of memory, it makes an unconditional request; the

free-space manager then lets the requesting task wait until sufficient memory is available. When a block of RAM is no longer needed, it may be sent to the reclamation exchange (RQFSRX) where it is merged into the free-space pool.

The building-block approach

The iRMX 88 Executive is completely configurable. The minimum "nucleus" consists of the following procedures: RQSTRT (initialization), RQCTSK (task-creation), RQCXCH (exchange-creation), RQSEND, and RQWAIT. Any of the remaining procedures are automatically included when they are referenced. Interrupt support and the system clock are separately configurable. The port addresses used for the 8259A programmable-interrupt controller, the 8251A programmable-communications interface, and the associated 8253 programmable-interval timer are configurable to any iAPX 86 or iAPX 88 port address. The transmission rate of the terminal handler is configurable from 150 to 19,200 baud. The interrupt levels for the system clock and terminal-handler interrupts are configurable to any level of the 8259A programmable-interrupt controller. Any unused level is configured to its default interrupt-service routines. The base of the interrupt vector used by the 8259A programmable-interrupt controller can be selected from interrupt levels 8 to 248. The smallest system time unit is 1 ms.

An "interactive configuration utility" (ICU 88) simplifies the configuration process. This utility, which executes on the Intel microcomputer-development system, asks the user a series of questions, allowing him to specify his own board configuration or to use a default that corresponds to the iSBC 86/12A, iSBC 86/05, or iSBC 88/40 boards.

To transfer an application from the iRMX 80 to the iRMX 88 Executive, the user must change the parameter for the create-task (RQCTSK) procedure from a 16-bit address to a 32-bit pointer. The description of the task can then be placed in random-access or read-only memory. In the iRMX 88, the definition of task has been expanded to incorporate a flag that indicates whether or not the 8087 numeric-data processor will be used. If that flag is set, an additional 94 bytes must be reserved for the task. Furthermore, the first parameter of the set-vector (RQSETV) procedure must be changed to a 32-bit pointer.

A robot demonstrates implementation

The iRMX 88 Executive demonstrates its value in a typical application: controlling an industrial robot's arm. The position of the arm in each of three dimensions is measured by an analog voltage on three input lines. The speed and direction of movement in each dimension is controlled with an analog

```

title ('display algorithm')
display:
DO:
  $include(:fl:ied.lit)
  $include(:fl:rainsd.ext)
  $include(:fl:rqundi.ext)
  $include(:fl:rquait.ext)
  $include(:fl:rasetv.ext)
  $include(:fl:rqlvl.ext)

DECLARE display$ied int$exchange$descriptor PUBLIC:

/* display interrupt service routine */
display$ir: PROCEDURE INTERRUPT 63:
  /* check for proper interrupt */
  OUTPUT(rq$8259a)=00h;
  IF NOT ((INPUT(rq$8259a) AND 80h) = 0) THEN
DO:
  /* output next character */
  OUTPUT(rq$8251) = output$buffer(output$index);
  output$index = output$index + 1;
  output$count = output$count + 1;
  IF output$count = 0 THEN
    CALL rq$send(.display$exchange);
  ELSE
    CALL rq$sendi(.display$exchange);
  END display$ir;

display: PROCEDURE:
  DECLARE message$address ADDRESS;
  CALL rq$setv(.display$ir,display$level);
  DO FOREVER:
    /* copy monitor and control information into the display
    buffer with the template of the display */
    .
    .
    .
    output$index = 1;
    output$count = LENGTH(output$buffer);
    /* enable the display interrupt and wait until the
    buffer has been output */
    CALL rq$setv(.display$ied);
    message$address = rq$wait(.display$ied,0);
  END display;

```

6. The PROCEDURE DISPLAY (bottom half) illustrates how the iRMX 88 operating system deals with random events. Here, interrupt level 63 has been chosen by the user.

```

input: PROCEDURE;
/* initialize */
DO FOREVER;
DO CASE field + 1;
/* next x position */
DO;
/* convert the character string in the input buffer
into a real number */
CALL rq$send(.control$x$ed,
.x$position$message);
END;
END input;
END input;

```

7. An endless loop is a typical method for capturing input. The skeleton code for PROCEDURE INPUT shows the interface with the executive via a CALL to an OS procedure.

```

$title ('monitor algorithm')
monitor:
DO;
/* monitor x task */
monitor$x: PROCEDURE;
DECLARE message$address ADDRESS;
CALL rq$exch(.waiting$ed);
DO FOREVER;
message$address = rq$wait(.waiting$ed,16);
OUTPUT(dac$command) = x$input$select;
message$address = rq$acpt(.waiting$ed);
x$control$input = SHR( INPUT(dac$low), 4) +
SHL( INPUT(dac$high), 4);
x$monitor$position = x$control$input * x$input$scale;
x$rate = ( x$position - x$monitor$position ) * 10;
x$position = x$monitor$position;
CALL rq$send(.x$action$ed,.x$position$message);
END;
END monitor$x;
END monitor;

```

8. PROCEDURE MONITOR \$X is a task that has to run every 16 ms. The IRMX 88 procedure RQWAIT accepts the desired time delay as an argument.

```

$title ('control algorithm')
control:
DO;
/* control x task */
control$x: PROCEDURE;
DECLARE message$address ADDRESS;
DO FOREVER;
message$address = rq$wait(.x$action$ed,0);
x$control$current = max$x$rate *
( next$x$position - x$position ) / ABS( next$x$position
- last$x$position );
x$control$output = SHL( FIX (
x$control$current * x$scale ), 4) + x$output$select;
OUTPUT(dac$high) = HIGH(x$control$output);
OUTPUT(dac$low) = LOW(x$control$output);
END;
END control$x;
END control;

```

9. This control algorithm sends data to the robot arm's X-axis through a d-a converter. The task remains dormant until the INPUT procedure from Fig. 7 makes it up.

current on three output lines.

The hardware consists of an iSBC 88/40 board with an iSBC 337 high-speed math multimodule, an iSBX 328 analog-output multimodule, and a CRT terminal with an RS-232 serial interface. The position of the arm in any dimension is read from the a-d converter on the iSBC 88/40; the converter is multiplexed over the three analog inputs. The d-a converter on the iSBX 328 analog-output multimodule is multiplexed to control the current to three stepper motors, which in turn control the arm's motion. A display (Fig. 5) is maintained on the CRT terminal and gives the robot's current position in three dimensions, the rate of change in each direction, and updatable fields for the desired position, as well as the maximum rate of change in each direction.

The display algorithm (Fig. 6) continuously updates the display from the control and monitor data bases. The monitor and control data bases update the display buffer. The display task initializes the display index and display count, enables the display interrupt, and waits at the display-interrupt exchange. The display-interrupt service routine responds to display interrupts by outputting the next character from the display buffer. When the last character in the display buffer has been output, a message is sent to the interrupt exchange; otherwise an end-of-interrupt is sent.

The input algorithm (Fig. 7) accepts digits as they are typed on the CRT terminal and enters them into the control field being updated. The fields on the right-hand side of the display are updated, starting with the top field and progressing to the bottom. Entry of a carriage return causes the values in the updated field to update the control-data base; the input algorithm then proceeds to the next field. Backspace discards the last digit typed. All other characters are ignored, and the user cannot type outside the defined fields. The input-interrupt routine receives a character with each interrupt and copies it into the display and input buffers.

The monitor algorithm (Fig. 8) samples the position of the robot's arm 60 times per second. It updates the monitor data base and then signals the control program through the appropriate exchange.

The control algorithm (Fig. 9) waits at the proper exchanges for a message from the input or the monitor algorithms. It then outputs a control current to the d-a converter based on the relative position of the robot's arm. □

iCSTM Products

3

3

ICS™ Products

Using Intel's Industrial Control Series in Control Applications

March 1979

Using Intel's Industrial Control Series in Control Applications

Peter Andersen
OEM Microcomputer Systems Applications

Using Intel's Industrial Control Series In Control Applications

Contents

INTRODUCTION	3-5
System Description	3-5
Control Algorithm	3-5
Basic System Configuration	3-6
WIRING INTERFACES	3-7
Analog Termination Panels	3-8
Low Voltage Digital Termination Panels	3-10
High Voltage Digital Termination Panels	3-13
Final Channel Assignments	3-16
SELECTING THE COMPUTER BOARDS	3-16
The Industrial Chassis	3-19
DETERMINATION OF SOFTWARE APPROACH	3-21
Assembler	3-22
PL/M	3-22
FORTRAN	3-23
BASIC	3-23
Final Selection of Language	3-23
DEFINING SOFTWARE TASKS	3-23
Oven Control Task	3-24
CRT Update Task Development	3-27
Parameter Update Task	3-28
Support Programs	3-29
FINAL IMPLEMENTATION	3-29
CONCLUSION	3-29
APPENDIX A — Selected Data Sheets	3-31
APPENDIX B — Ladder Diagram of System	3-38
APPENDIX C — Program Source Listings ..	3-39

I. INTRODUCTION

The introduction of the single board computer as a tool for the system designer has opened the way for many varied application areas to benefit from the advantages of computer utilization. A problem still exists, however, because the available I/O configurations have been largely incompatible with the wiring and packaging techniques required in industrial environments. This problem is overcome by the utilization of the Intel® iCST™ product family. The purpose of this application note is to provide a representative approach to the implementation of a computerized solution to an industrial control system.

System Description

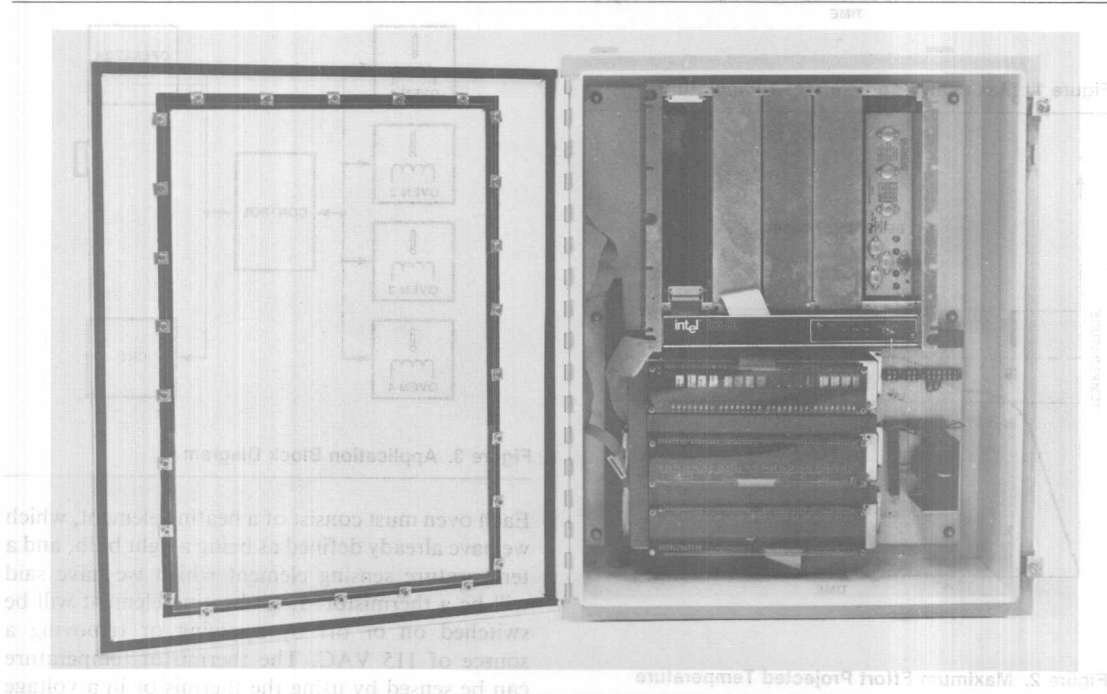
This application note will deal with a control system which will regulate the temperature in each of four ovens. Each oven will be defined as utilizing a light bulb for heating. Normal convection will be used to provide cooling. The internal temperature will be measured by means of a thermistor installed in each oven. We will assume that we will be required to implement some type of operator panel near the ovens which will allow the status of each oven to be monitored. This approach is similar to many common industrial applications which require a supervisory control station in one area and a separate operator interaction panel near the

equipment being controlled. The setpoint and tolerances should be input from an external location.

With these facts about our system defined, we can begin a step by step solution to providing a computerized control system to operate the ovens. We will discuss the various equipment trade-offs and the decisions which will be used to define the hardware/software designs.

Control Algorithm

Before we can begin the design of our system, we must have a clear idea of the technique we will use to control the system. Our control system must maintain the oven temperature within a predefined and fairly narrow range of the setpoint. Let us make an assumption that the light bulb will be controlled digitally, meaning that the bulb must either be turned fully on or it must be turned fully off. The obvious control technique then becomes turning the bulb on when the temperature of the oven is below our lower limit and turning the bulb off when the temperature is above the higher limit. It seems reasonable to assume that this technique will provide a temperature in the oven which varies sinusoidally with time. This is true because even though the lamp is turned off, it will continue to generate heat for a short period of time. Likewise, when the bulb is turned on, it will not instantly be able to provide heat to raise the temperature of the



chamber. We would expect to have a system response such as is shown in Figure 1. A better method of control can be devised if we provide some type of temperature prediction into our control algorithm. Since this utilizes the rate of temperature increase or decrease, it will involve a type of derivative control system. This derivative control action will tend to dampen the temperature oscillations which might be encountered if only an instantaneous on-off control system were utilized. Figure 2 shows the response with time that we might expect with this type of control system.

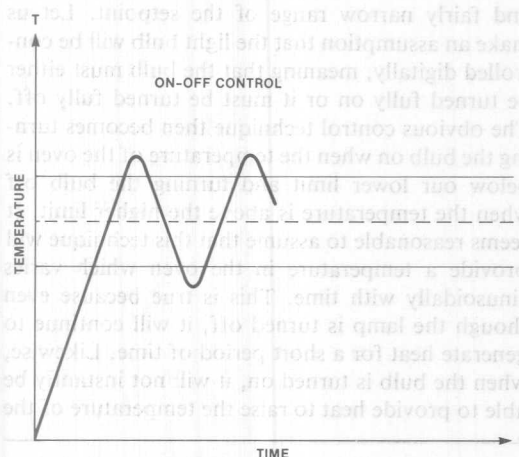


Figure 1. Maximum Effort Current Temperature

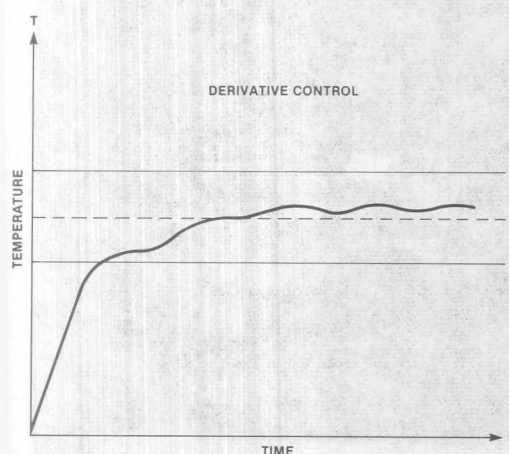


Figure 2. Maximum Effort Projected Temperature

The second approach is superior to the first because the control will provide a much smaller oscillation of the oven temperature. Other solutions are possible, such as providing a modulated output to the lamp. However, in an attempt to provide a simple model upon which to expand our system solution, we will assume that the second approach will provide us with an accurate enough control of the oven temperature.

Having made the decision as to the control technique, we can proceed with the task of determining the general system configuration. That is, we can define the physical system characteristics and the components to which we must interface the computer system. This approach is identical to that which would be used in a conventional control system design.

Basic System Configuration

Based upon the data which we have provided so far, it is possible to build a block diagram of the system's major components. The system consists of four ovens, an operator's panel, a data entry panel, and the actual control logic. A block diagram for the system is shown in Figure 3. We must now further define the elements which make up each of these blocks.

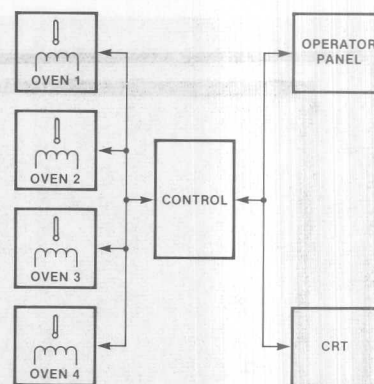


Figure 3. Application Block Diagram

Each oven must consist of a heating element, which we have already defined as being a light bulb, and a temperature sensing element which we have said will be a thermistor. Each heating element will be switched on or off by applying or removing a source of 115 VAC. The thermistor temperature can be sensed by using the thermistor in a voltage

divider circuit. We can then measure the voltage across a fixed resistor to obtain an analog signal which is proportional to the oven temperature. We will determine the required value of the fixed resistor at a later time.

The operator's panel should be designed to provide the workflow operator with basic information as to the status of each oven. It should also allow some method by which he can inhibit the operation of any oven should it become necessary for charging or servicing the oven. We can then define the basic elements which should make up the operator's control panel. Each oven should have associated with it the following controls and indicators:

1. Oven ON/OFF Switch — This switch will allow the operator to inhibit the oven operation by turning the appropriate oven switch to OFF.
2. Oven RUNNING Indicator — This indicator will provide a visual indication that the oven is activated and that the temperature is being controlled.
3. Oven IN TOLERANCE Indicator — This indicator will turn on when the oven temperature falls within the allowable bandwidth around the setpoint for that oven.
4. Oven ALARM Indicator — This indicator is the complement of the in tolerance lamp. It will be turned on when the oven is activated and the temperature does not lie within the desired bandwidth.
5. Oven CAUTION Indicator — It may be necessary to alert the operator to a potential oven temperature control problem before it actually occurs and sets off the alarm indicator. Since we have defined our control algorithm as utilizing a type of derivative control, we can project the oven temperature ahead in time. We will turn the oven caution indicator on when we predict that the oven temperature will lie outside of the desired bandwidth in a predetermined future time period.

We have now defined the operator interface which we will utilize to control and monitor the oven processes.

At this point, we will make a decision that the interface used to input the setpoints will utilize a CRT terminal. Though the decision may seem to be completely arbitrary, we will see later that CRT terminals provide an extremely useful device for allowing an operator to communicate with the system. Once the decision has been made, we have no

further requirements to consider hardware design for this terminal, as the entire operation can be handled in the software development which will be considered later.

A common technique for documenting a system is the ladder diagram. At this time, we can construct a ladder for our control system. Unlike conventional design techniques, our ladder diagram need only be concerned with the actual drive and sensing circuits since the logic required to drive the various outputs will be defined using software. This results in a considerable simplification of the design process. A ladder diagram for a typical oven is shown in Figure 4. We can defer the implementation of the control algorithm until we begin to develop the software portion of our control system. It is now possible to complete the external hardware design and to implement the system wiring package.

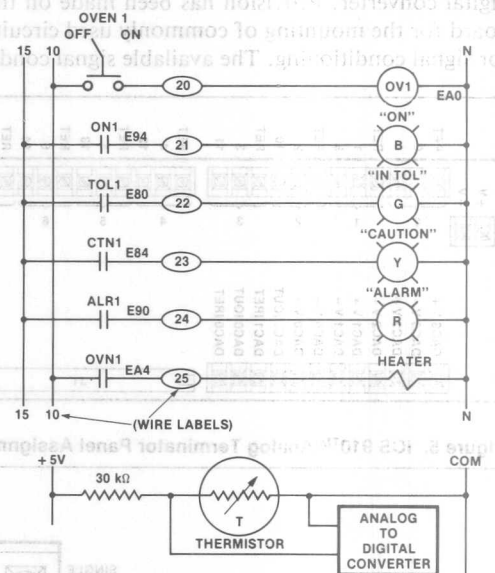


Figure 4. Ladder Diagram of One Oven

II. WIRING INTERFACES

A major pitfall in utilizing a computer for control systems has traditionally been the requirement for the design engineer to expend a considerable amount of his time in designing interfaces to connect the physical wiring to the computer system. The introduction of Intel's product line of termination panels has essentially eliminated the require-

ment of designing interfaces and allows more engineering time to be spent providing a solution to the application. Before we continue with the specific design, we should spend some time discussing the various types of termination panels available and the general characteristics of each panel.

Analog Termination Panels

The Intel® iCS 910™ Analog Termination Panel has been designed to provide a simple means of terminating the analog wiring and of providing an interface to the control system input/output. All wiring is terminated utilizing pressure type screw barrier blocks. Termination blocks have been provided to allow the termination of up to 32 single-ended or 16 differential channels of analog input. For use in a differential input environment, such as we will be using, the terminator blocks provide wiring terminations compatible with shielded cable inputs in that provision has been made to accept the shield of each input signal. The shield is then carried through the on-board circuits to the analog-to-digital converter. Provision has been made on the board for the mounting of commonly used circuits for signal conditioning. The available signal condi-

tioning circuits provide for installation of current termination resistors and the installation of a single pole low pass filter network. The basic barrier assignments for the iCS 910 termination panel are shown in Figure 5. The possible circuit networks for this panel are illustrated in Figure 6. A complete description of the analog termination panel can be found in the *iCS 910 Analog Signal Conditioning/Termination Panel Hardware Reference Manual* (manual order number 9800800A).

The functions of the analog termination panel will become more clear as we develop the actual configuration required to support our oven application. Referring to the ladder diagram (Figure 4) we see that a fixed resistor is necessary to provide the voltage divider network to sense the oven temperature. The current termination resistor (R_c) on the iCS 910 board can be used to provide a convenient mounting location for this component (refer to iCS 910 circuit schematic, Figure 6). At this point, we must make a design decision regarding the utilization of a low pass filter for our analog circuits. Since the oven temperatures are not expected to exhibit rapid fluctuations with time, the use of a low pass filter will not adversely effect the temperature

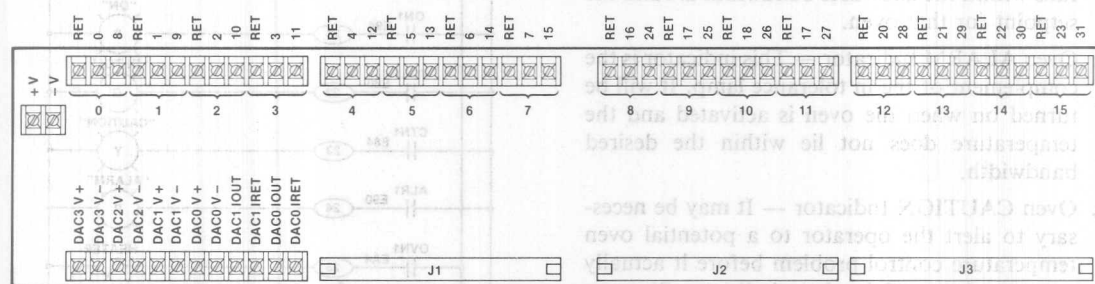


Figure 5. iCS 910™ Analog Terminator Panel Assignments

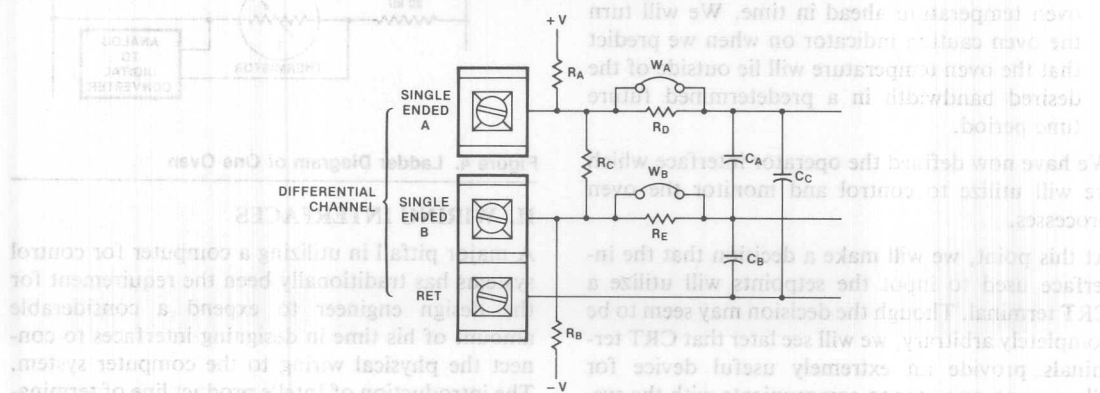


Figure 6. Typical Circuit on Analog Terminator

The graph plots the ratio V_{AD}/V_{max} on the y-axis against FREQUENCY (Hz) on the x-axis. The y-axis ranges from 0.0 to 1.0 with increments of 0.2. The x-axis is logarithmic, with major ticks at 0.1, 1, 10, 100, and 1000 Hz. The curve is a high-pass filter response, remaining at 1.0 until approximately 0.1 Hz, then rolling off at a rate of -6 dB/OCTAVE.

FREQUENCY (Hz)	V_{AD}/V_{max}
0.05	1.0
0.1	1.0
0.2	0.98
0.5	0.89
1.0	0.79
2.0	0.63
5.0	0.39
10.0	0.25
100.0	0.025
1000.0	0.0025

is continued for a full 24 hr period. In the

1. TO THERMISTOR #1

2. TO THERMISTOR #2

3. TO THERMISTOR #3

4. TO THERMISTOR #4





...the various indicators

colours were used to drive the external indicators. The 920™ provides us with a solid state device to perform the same function, the optical isolator. We can use these devices to provide a highly reliable and low cost alternative to the relay circuit. The new ICS 920™ Digital Signal Converter, Terminal Panel provides us with a convenient vehicle for mounting the optical isolator circuit and for terminating the wiring.

tem and the physical devices which are to be monitored or controlled. Figure 10 shows the placement of the components onto the board.

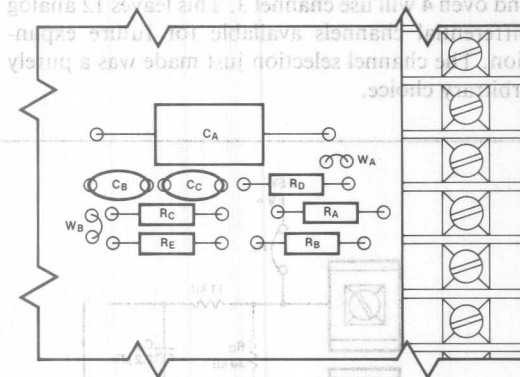


Figure 10. Analog Terminator Component Locations

Low Voltage Digital Termination Panels

Looking again at our ladder diagram for an oven control system (Figure 4), we see the need to provide a second type of interface signal. This is to provide the switching for the various indicator lamps used on the operator's control panel. Traditionally, this interface has been handled by using electromechanical relays. The coils would be driven by the low voltage control system and the relay contacts were used to drive the external indicators. Modern technology provides us with a solid state device to perform the same function, the optical isolator. We can use these devices to provide a highly reliable and low cost alternative to the relay interface. The Intel® iCS 920™ Digital Signal Conditioning/Terminator Panel provides us with a convenient vehicle for mounting the optical isolator circuits and for terminating the wiring associated with the indicator devices.

The iCS 920 panel is designed to be used by those interface circuits which incorporate operating voltages less than 50 volts and which generally use currents which are smaller than 300 mA. These limits are given only for a general guideline since a wide variety of optical isolators and drivers are available for use on the board. Some of the devices are capable of handling greater voltages or currents. A representative list of available devices and complete details of the termination panel are available in the *iCS 920 Digital Signal Conditioning/Termination Panel Hardware Reference Manual* (manual order number 9800801A).

The digital panel provides terminations for up to 24 digital channels, each of which can be configured as either an input or an output channel according to the specific application requirements. As with the analog termination panel, all wire terminations are made using pressure type barrier strips which will accept up to 16 gauge wire. The 24 digital channels correspond with those input/output channels assigned to the standard Intel I/O configurations used on the single board computers and I/O expansion boards. We will dwell more on this subject later when we define the addresses associated with each circuit which we desire to incorporate into the termination panel.

Since the digital channels can be configured into either an input or an output mode, it is wise to discuss each configuration so that a clear understanding of the board can be obtained, even though our application example will only use the output mode with this board.

Figure 11 provides a schematic of the panel when it is configured for a digital input mode. To set up a channel to operate as an input, it is necessary to add at least two jumpers to the wire-wrap jumper posts. As can be seen, pins 6 and 4 must be connected together as well as pins 3 and 5. If the board is to provide a visual LED indication of the channel status, an additional jumper should be installed between pins 1 and 2 of the jumper posts. If this is done, be certain to take into account the additional current requirements when calculating the required input resistors. Two resistor mounting locations are provided to allow installation of selected components to handle the current limit through the optical isolator (Rx) and the threshold voltage for turn-on of the device (Ry). A complete and detailed procedure for selecting these resistors based upon the input voltages is provided in the iCS 920 hardware reference manual mentioned earlier. Provision has also been made on the termination panel for the installation of a diode (CR) to protect against reverse bias application.

The components have been placed on the board arranged in groups of two channels. This eases the task of finding various components or of locating the holes for installing the required components. This layout is illustrated in Figure 12. It is important to take note of the physical placement of the optical isolator chips in the 20-pin socket. This installation location must be followed rigorously when using a channel in an input mode. Also take note that provisions are provided for mounting two sizes of resistors in location (Rx). This will accom-

modate the power dissipation requirements which will be encountered in various application situations. Referring again to Figure 12, note that the upper half of the layout represents odd channels and the lower portion of the layout is used for even channel component mounting.

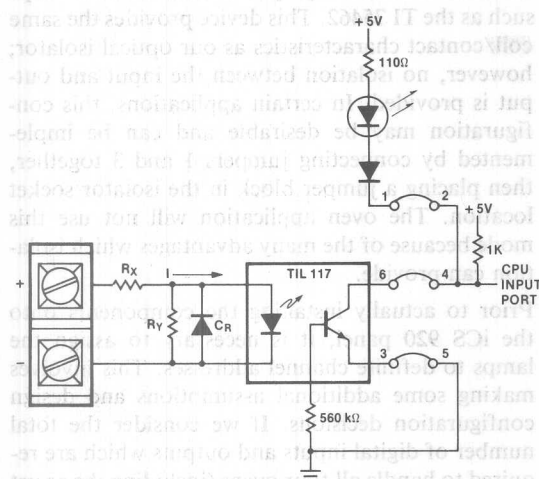


Figure 11. iCS 920™ Digital Terminator Input Configuration

When the iCS 920 panel is used in this input mode, it corresponds to the utilization of a relay coil to sense some external contact closure. The resistors can be thought of as selecting the coil's operating voltage and the diode provides the same transient

protection function as when installed on an electro-mechanical relay. Finally, the optical isolator output corresponds to the contacts associated with the relay coil. As we will see later, this approach provides us with an unlimited number of contacts per relay coil.

The oven application requires a contact for driving the indicator lamps associated with each oven. If we define the driving voltage to be 24 volts DC, we will find that the voltage and current requirements fall within the limits specified for using the iCS 920 Digital Signal Conditioning/Termination Panel. Let us examine in more detail how this can be accomplished.

We will select an industrial indicator assembly which utilizes a full voltage 24-volt lamp. Typical lamps would be type 387. This will require a drive of 40 mA at 28 volts. Our switching device must be capable of driving this load. The analogy used earlier to compare the optical isolator with a relay in an input mode holds true when we utilize the devices in an output configuration. If we examine the data sheet for the current switching characteristics of a typical optical isolator, say the TIL 113 (Appendix A), we can see that the current and voltage requirements fall well within the allowable ratings of the device. We have selected the relay contact characteristics! We need not concern ourselves with the selection of current limitation resistors (coil voltage ratings) since this circuitry is provided on the terminator panel when a circuit is

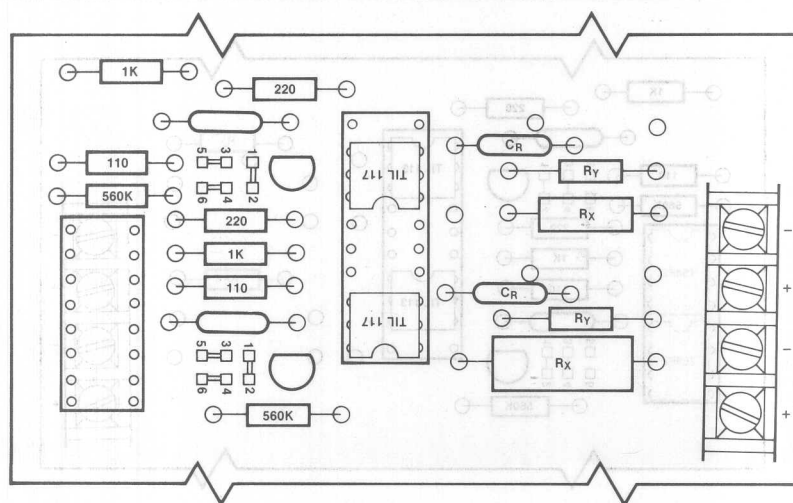


Figure 12. Digital Terminator Input Parts Layout

configured in an output mode. If we refer to Figure 13, we can see the on-board schematic for the output drive mode of operation. Two jumpers must be installed for each output channel. The first, between pins 1 and 2, is used to enable the LED channel status indicator. The second, between pins 3 and 4, actually connects the computer generated drive signal to the input of the optical isolator (analogous to connecting the relay coil to the driving line). Provision has been made on the circuit board for only one optional component in the output mode; this is the resistor (R_z). This component has the effect of increasing the response time of the switching device. Because our indicator lamps are not time critical, we will choose to omit the installation of this component.

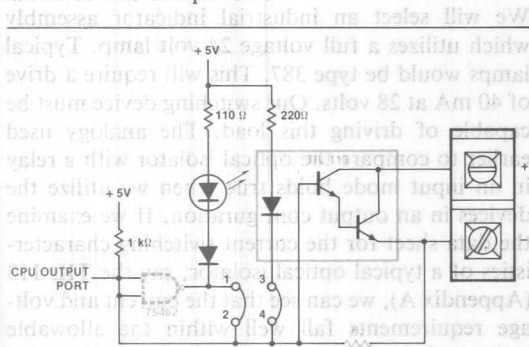


Figure 13. iCS 920™ Digital Terminator Output Circuit

Figure 14 provides a drawing showing the location of the components on the iCS 920 panel when it is utilized as an output switch. Again note the place-

ment of the optical isolators in the 20-pin sockets. Also note the jumper arrangement used to provide the required output circuitry.

Again referring to Figure 13, we see that an alternative to using the optical isolator for a switch exists. Provision has been made on the panel for the installation of high power buffer/driver chips such as the TI 75462. This device provides the same coil/contact characteristics as our optical isolator; however, no isolation between the input and output is provided. In certain applications, this configuration may be desirable and can be implemented by connecting jumpers 1 and 3 together, then placing a jumper block in the isolator socket location. The oven application will not use this mode because of the many advantages which isolation can provide.

Prior to actually installing the components onto the iCS 920 panel, it is necessary to assign the lamps to definite channel addresses. This involves making some additional assumptions and design configuration decisions. If we consider the total number of digital inputs and outputs which are required to handle all four ovens (including the as yet unconsidered switch and heater signals), we see that a total of 24 channels will be required. These will be broken out as shown below:

No. of Channels	Type	Function
16	DC	Oven indicator lamps
4	AC	Oven heaters
4	AC	Oven RUN switches

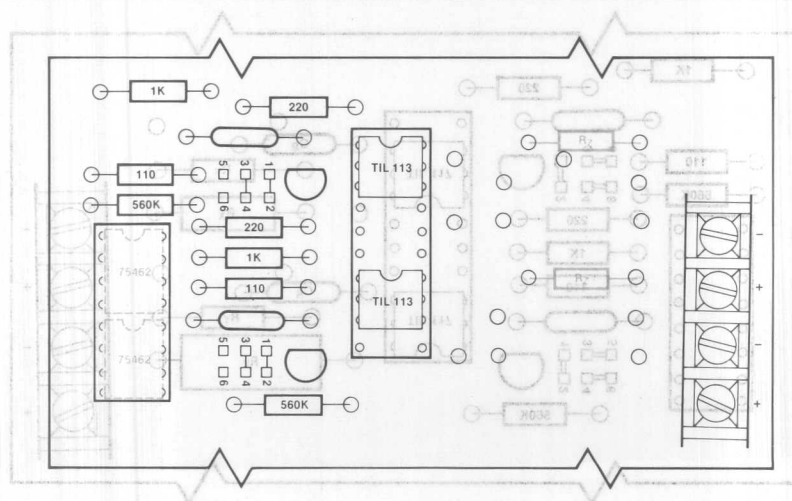


Figure 14. Digital Terminator Output Configuration

We have indicated that the 16 indicator lamps can be handled using the iCS 920 panel. An examination of the data sheets for the various Intel single board computers and expansion boards provides us with the fact that a common characteristic of most boards is the use of at least one Intel 8255 Programmable Peripheral Interface. This provides us with at least 24 I/O lines with which to work on each single board computer. We can then assume that we will not require an I/O expansion board to implement our application. Ideally, we can handle our total requirements with one parallel interface.

The various Intel parallel ports are brought off of the computer and expansion boards using edge connectors. These edge connectors are then connected to the termination panels using a standard ribbon cable assembly, effectively providing an extension of the I/O ports out to the termination panels. The 24 channels are grouped into three I/O ports (each consisting of 8 channels or bits) which are then called port A, port B, and port C. When connected to the iCS 920 panel, these ports and their bit assignments will be as shown in Figure 15.

At this point, we seem to be in a dilemma since we would like to use all 24 channels and we have used only 16 of them on our panel while we have utilized the edge connector of the interface. It would be desirable to have some technique to extend the other 8 channels to a high voltage terminator panel. It might be well to interrupt our channel assignments at this time to jump ahead and consider the features of the iCS product line which will enable us to accomplish our interface desires. We will then consider the interface of the high voltage signals to our control system before returning to the problem of assigning port locations to our lines.

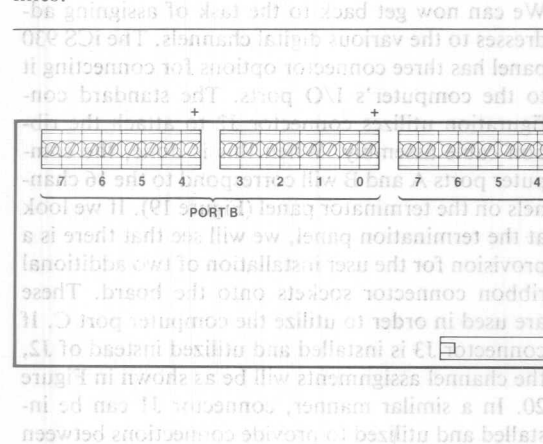


Figure 15. iCS 920™ Digital Terminator Port Assignment

High Voltage Digital Termination Panels

The Intel® iCS 930™ AC Signal Conditioning/Termination Panel is designed to interface up to 16 AC signals (up to 280 volts at 3 amps) or high current DC signals (up to 50 volts at 3 amps) to the parallel ports of the Intel single board computers or I/O expansion modules. The barrier strip terminations on this panel are designed to easily handle the 14 gauge wire commonly found in applications requiring the use of the AC terminator.

Solid state relays are used to provide the interface between the computer I/O ports and the physical plant devices. These devices make the utilization of the panel a simple task once a ladder diagram of the required circuits has been drawn. As we have previously mentioned and as is clear from looking at Figure 4, we shall need to utilize eight of the available circuits, four for input and four for output. The implementation of each signal type requires only that we insert the correct type of solid state relay into the appropriate socket.

First, consider the input configuration which is required to sense the position of the oven RUN switches. Figure 16 shows the circuit schematic when used in the input mode. We can see that the output signal will turn on when the input power is applied. Like the digital termination panel, each circuit's status is indicated by means of an LED indicator installed on the board. The input circuit is protected by a socketed 3-amp fuse which may be replaced without the need to solder any components. The solid state relay used for this configuration should be a type IAC5 which is available from either Opto-22 or Motorola. Complete details of available relays and their uses on the board are available in the *iCS 930 AC Signal Conditioning/*

Termination Panel Hardware Reference Manual (manual order number 9800802A). Keep in mind the fact that although this application note represents the solid state relays as being actual relays and contacts, they in fact are solid state and contain no moving parts.

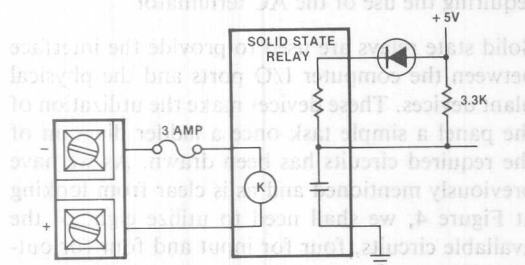


Figure 16. iCS 930™ AC Terminator Input Circuit

The output configuration is utilized to turn the heater elements (the light bulbs) on and off. Figure 17 provides us with a schematic of the output circuitry. In this case, we will insert a solid state relay of type OAC5 which will handle up to 140 volts RMS at 3 amps. In some cases, it might be desirable to add certain components to the terminator panel when using it in the output mode. Two possible circuit configurations are possible. The first and perhaps the most common will consist of installing a MOV (metal oxide varistor) across the solid state relay contacts. This will be required when the load being driven is inductive in order to prevent the transients generated by the load from damaging the triac in the SSR (solid state relay). Since the SSRs utilize zero voltage switching and the load in our ovens is resistive rather than inductive, our application will not necessitate the installation of this device. The second possibility for additional circuitry also involves driving inductive loads. When the load is highly inductive, a possibility exists that reliable operation of the SSR may not occur because of incorrect values for the dv/dt (a complete description of this phenomenon is available in various publications available from the manufacturers of the solid state relay devices). Provision has been made for installation of an external snubber network should this be required. Again, our oven control system will not require this type of circuitry. Figure 18 is provided for reference should the reader desire to see the location of the additional components on the panel. It should be noted that the component placement does not

allow the installation of the MOV and the snubber simultaneously.

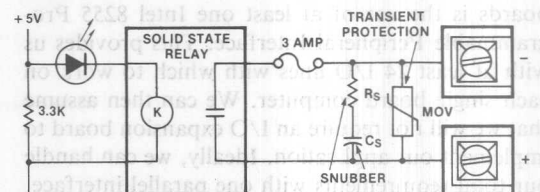


Figure 17. iCS 930™ AC Terminator Output Circuit

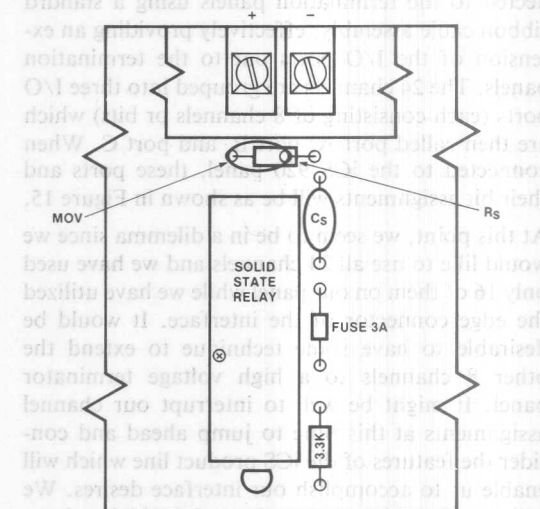


Figure 18. AC Terminator Component Locations

We can now get back to the task of assigning addresses to the various digital channels. The iCS 930 panel has three connector options for connecting it to the computer's I/O ports. The standard configuration utilizes connector J2 to attach the ribbon cable assembly. When this is done, the computer ports A and B will correspond to the 16 channels on the terminator panel (Figure 19). If we look at the termination panel, we will see that there is a provision for the user installation of two additional ribbon connector sockets onto the board. These are used in order to utilize the computer port C. If connector J3 is installed and utilized instead of J2, the channel assignments will be as shown in Figure 20. In a similar manner, connector J1 can be installed and utilized to provide connections between the computer port C and the other eight SSR positions. If we choose the 16 lines required for driving

the indicator lamps from the iCS 920 panel to be ports A and B, then it seems reasonable to assign the eight remaining lines required on the iCS 930 to port C. A feature of utilizing standard ribbon cable assemblies is the ability to easily add ribbon plug connectors to the cable. This will result in an assembly transferring ports A, B and C to the iCS 920 panel (however, port C is not used) and which continues the port C signals to the iCS 930 panel.

Individual channel assignments can now be made, grouping the inputs and outputs together in groups of four (this is done because of a requirement of the single board computers to share terminator and driver component packages in groups of four). Figure 21 provides a drawing showing the channel assignments and the physical wiring locations which will be used to connect the oven heaters and switches.

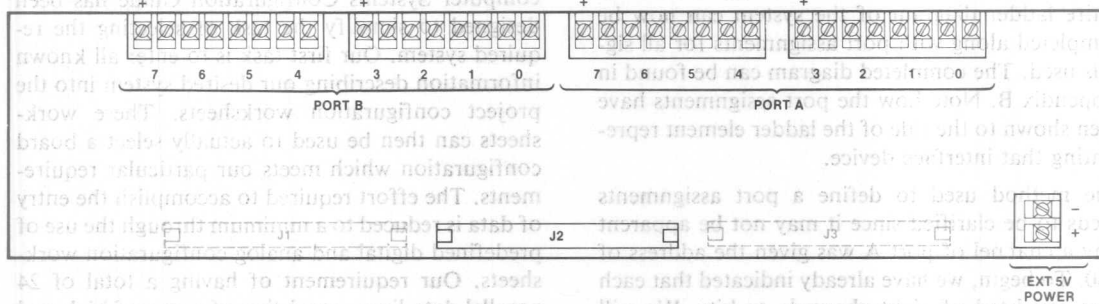


Figure 19. iCS 930™ AC Terminator Port Assignments

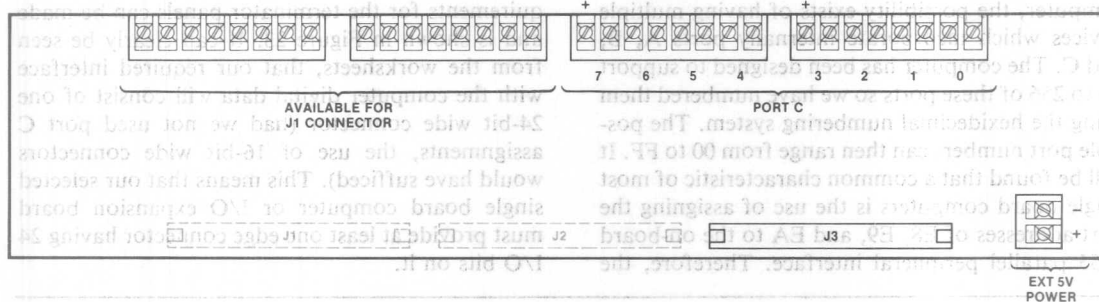


Figure 20. iCS 930™ AC Terminator Port Assignments

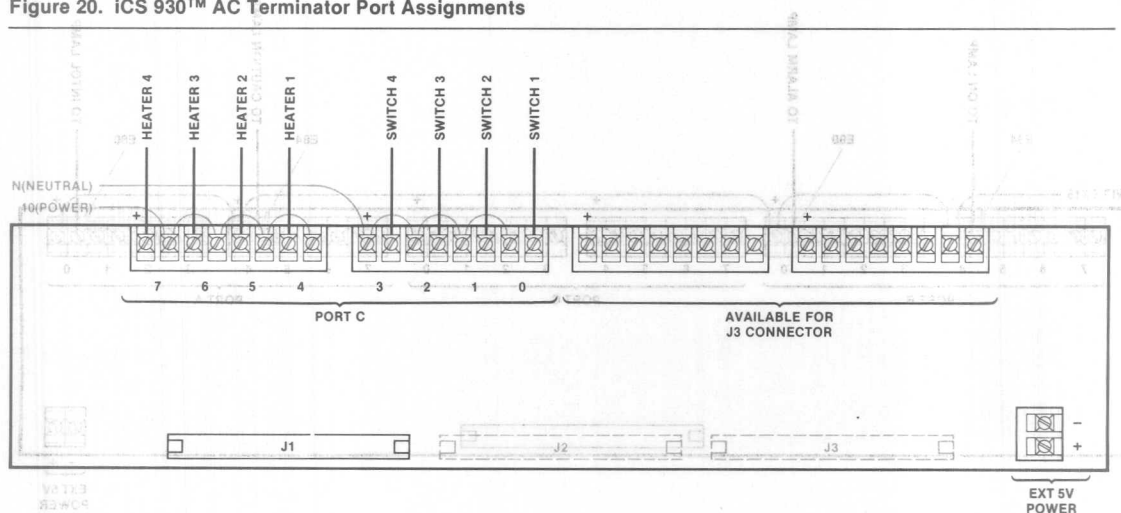


Figure 21. iCS 930™ AC Terminator Application Configuration

Final Channel Assignments

The only task remaining before we have completed our task of assigning channel numbers and physical wire and component locations is to assign these channels on the iCS 920 digital termination panel. Since we have already determined that we will utilize ports A and B, this becomes a simple matter, requiring only an arbitrary assignment of lamp locations using these port bits. The assignments made for one oven can be seen in Figure 22. The entire ladder diagram of the system can now be completed along with port assignments for all signals used. The completed diagram can be found in Appendix B. Note how the port assignments have been shown to the side of the ladder element representing that interface device.

The method used to define a port assignments needs to be clarified since it may not be apparent why a channel of port A was given the address of E80. To begin, we have already indicated that each port consisted of eight channels or bits. We will number these bits from 0 to 7. Since it is possible to have many input/output devices connected to the computer, the possibility exists of having multiple devices which incorporate internally ports A, B, and C. The computer has been designed to support up to 256 of these ports so we have numbered them using the hexadecimal numbering system. The possible port numbers can then range from 00 to FF. It will be found that a common characteristic of most single board computers is the use of assigning the port addresses of E8, E9, and EA to the on-board 8255 parallel peripheral interface. Therefore, the

first channel of port A would be defined as having an address of E80; the second channel of port B would be E91, and so forth.

III. SELECTING THE COMPUTER BOARDS

To this point we have delayed the selection of the boards which will be required to provide the computerized control system. The Intel OEM Micro-computer Systems Configuration Guide has been designed to simplify the task of selecting the required system. Our first task is to enter all known information describing our desired system into the project configuration worksheets. These worksheets can then be used to actually select a board configuration which meets our particular requirements. The effort required to accomplish the entry of data is reduced to a minimum through the use of predefined digital and analog configuration worksheets. Our requirement of having a total of 24 parallel data lines, consisting of a mix of high and low level interfaces, can be met by the 24-bit AC/DC combination. Our assignments of requirements for the terminator panels can be made and is shown in Figure 23. It can clearly be seen from the worksheets, that our required interface with the computer digital data will consist of one 24-bit wide connector (had we not used port C assignments, the use of 16-bit wide connectors would have sufficed). This means that our selected single board computer or I/O expansion board must provide at least one edge connector having 24 I/O bits on it.

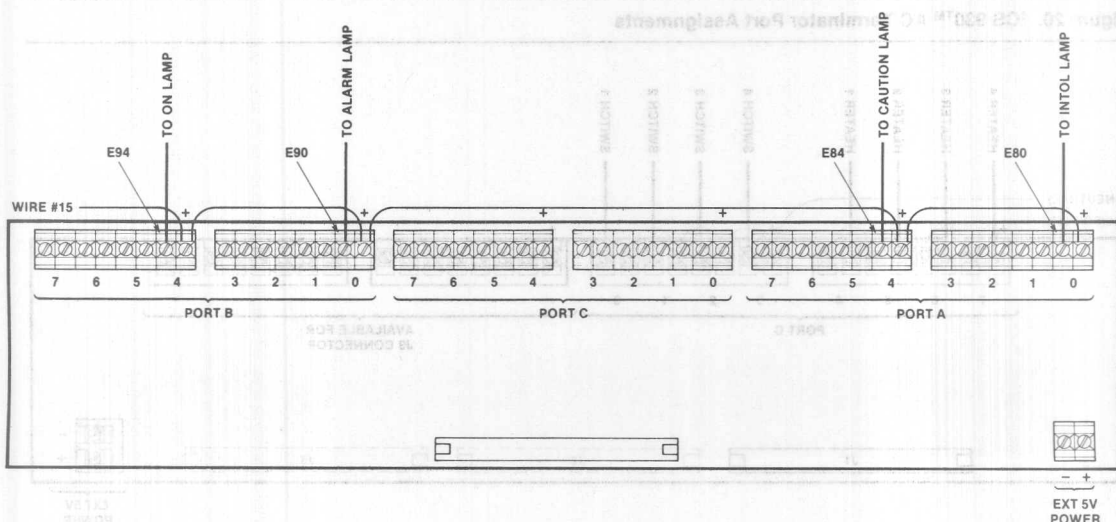


Figure 22. Digital Panel Application Configuration

DIGITAL CONFIGURATION WORKSHEET

PROJECT

This worksheet will provide the required digital interface configuration data which is required to complete the Project Configuration Worksheet.

Enter Number of Channels

Enter # of Discrete AC Outputs (115-230 VAC) 4 (A)
 Enter # of Discrete AC Inputs (115-230 VAC) 4 (B)
 Enter # of Discrete DC Outputs (Current > 300 MA) 0 (C)
 Enter # of Discrete DC Outputs (Current < 300 MA) 16 (D)
 Enter # of Discrete DC Inputs 0 (E)

Compute the Number of iCS 920™ and iCS 930™ Termination Panels

First compute the number of Parallel I/O ports (8-bits each port) required on your iSBC™ board. Round all computations up to the nearest whole integer unless instructed otherwise!

Compute # of iCS 930 Interface Output Ports ((A+C)/8) 1 (F)
 Compute # of iCS 930 Interface Input Ports (B/8) 1 (G)
 Compute # of iCS 930 Termination Panels ((F+G)/2) 1 (H)
 Compute # of iCS 920 Interface Output Ports (D/8) 2 (J)
 Compute # of iCS 920 Interface Input Ports (E/8) 0 (K)
 Compute # of iCS 920 Termination Panels ((J+K)/3) 1 (L)

Optimization of Digital I/O Port Usage for Minimum I/O Configuration

Compute # of iCS 930 Output "Overflow Channels" (DO NOT ROUND OFF)
 (A+C)/8 QUOTIENT 0 (M)
 (Overflow Channels) REMAINDER 4 (N)
 Compute # of iCS 930 Input Overflow Channels (DO NOT ROUND OFF)
 (B/8) QUOTIENT 0 (P)
 REMAINDER 4 (R)
 Compute # of iCS 920 Output Overflow Channels (DO NOT ROUND OFF)
 (D/8) QUOTIENT 2 (S)
 REMAINDER 0 (T)
 Compute # of iCS 920 Input Overflow Channels (DO NOT ROUND OFF)
 (E/8) QUOTIENT 0 (V)
 REMAINDER 0 (W)
 Compute 8-Bit Input Ports Required (P+V) 0 (X)
 Compute 8-Bit Output Ports Required (M+S) 2 (Y)
 Compute 4-Bit Output Ports Required ((N+T)/4) (ROUND UP) 1 (Z)
 Compute 4-Bit Input Ports Required ((R+W)/4) (ROUND UP) 1 (AA)
 Compute 8-Bit Port C Requirements ((Z+AA)/2) (ROUND UP) 1 (BB)
 Total I/O Parallel Ports Required (X+Y+BB) 3 (CC)
 Total # of 24 Channel Parallel I/O iSBC Board Edge Connectors
 (CC/3) (ROUND UP TO INTEGER) 1 (DD)

Compute Power Requirements for the Termination Boards (DO NOT ROUND OFF)

Compute +5V for iCS 920 Board Outputs (.061×D) 976 (EE)
 Compute +5V for iCS 920 Board Inputs (.023×E) 0 (FF)
 Compute +5V for iCS 930 Board Outputs (.020×(A+C)) 080 (GG)
 Compute +5V for iCS 930 Board Inputs (.012×B) 048 (HH)
 Compute iCS 920 Power Requirements (EE+FF) 976 (JJ)
 Compute iCS 930 Power Requirements (GG+HH) 123 (KK)

Enter the appropriate data into the Project Configuration Worksheet as shown below:

SEE INSTRUCTION SHEET

PROJECT CONFIGURATION WORKSHEET

EQUIPMENT PARAMETERS:

YOUR REQUIREMENTS	PRODUCT	MEMORY		SERIAL I/O		PARALLEL INPUT/OUTPUT		ANALOG INPUT/OUTPUT		POWER REQUIREMENTS				COST			
		RAM (KB)	EPROM (KB)	RTS	RS232C	Parallel Lines	Ports	Inputs	Outputs	5V	12V	5V	12V	Line	Qty	Qty	
(L) ea iCS-920						(E)	(D)										
(H) ea iCS-930						(B)	(AC)	(DD)									
TOTAL																	
INTEL SOLUTION	SLOT #																

Figure 23. Digital Configuration Worksheet

The required power requirements of the termination panels can be calculated using the data provided in the digital configuration worksheet. The information regarding the necessary connectors and the power requirements should then be transferred to the project configuration worksheet (Figure 24).

PROJECT CONFIGURATION WORKSHEET

EQUIPMENT PARAMETERS:

NAME	TYPE	NO. OF CHANNELS	NO. OF ANALOG CHANNELS	NO. OF DIGITAL CHANNELS	NO. OF I/O CHANNELS	NO. OF POWER CHANNELS
100. ICS-150			0	16		.5%
101. ICS-150			4	4		.1%
102. ICS-150						
103. ICS-150						
104. ICS-150						
105. ICS-150						
106. ICS-150						
107. ICS-150						
108. ICS-150						
109. ICS-150						
110. ICS-150						

Figure 24.

A similar technique is used to configure the analog signals using the standard analog configuration worksheet as shown in Figure 25. It can be seen that our application will require a single cable connection to a differential input edge connector of an analog input board. The power requirements can be calculated from the current requirements to drive the thermistors and the sensing resistors. The data is entered into the appropriate columns of the configuration tables and then transferred to the project configuration worksheet.

ANALOG CONFIGURATION WORKSHEET

PROJECT QUEUE CONTROLLER

This worksheet will provide the required analog interface configuration data which is required to complete the Project Configuration Worksheet.

Enter Number of Channels

Enter # of Single Ended High Level Analog Channels 0 (A)
 Enter # of Differential High Level Analog Channels 0 (B)
 Enter # of Differential Low Level Analog Channels 0 (C)
 Enter # of Analog Output Voltage Channels 0 (D)
 Enter # of Analog Output Current Channels 0 (E)

Compute the Number of ISBC™ Board Edge Connectors

Unless otherwise noted, round all computations to the next largest integer!

Compute # of High Level Single Ended Analog Connectors (A/16) 0 (F)
 Compute # of High Level Differential Connectors (B/8) 0 (G)
 Compute # of Low Level Differential Connectors (C/8) 0 (H)
 Compute # of Analog Interface Input Connectors (F+G+H) 0 (J)

Compute the Number of ICS-910™ Termination Panels

Enter Analog Out Connectors (D/4+E/2) 0 (K)
 Enter # of Analog In Connectors (J/2) 0 (L)
 Enter Larger of (K) or (L) 0 (M)

Place the appropriate data into the Project Configuration Worksheet as shown below:

PROJECT CONFIGURATION WORKSHEET

EQUIPMENT PARAMETERS:

NAME	TYPE	NO. OF CHANNELS	NO. OF ANALOG CHANNELS	NO. OF DIGITAL CHANNELS	NO. OF I/O CHANNELS	NO. OF POWER CHANNELS
100. ICS-150			0	16		.5%
101. ICS-150			4	4		.1%
102. ICS-150						
103. ICS-150						
104. ICS-150						
105. ICS-150						
106. ICS-150						
107. ICS-150						
108. ICS-150						
109. ICS-150						
110. ICS-150						

Figure 25.

The only remaining physical element of our control system which we have not defined is the CRT terminal which will be used for setpoint entry and modification. Communications with a terminal requires that we provide a serial RS232C port in our control system. This port requirement is entered

onto the worksheet and the system requirements are totaled as shown in Figure 26.

PROJECT CONFIGURATION WORKSHEET

EQUIPMENT PARAMETERS:

NAME	TYPE	NO. OF CHANNELS	NO. OF ANALOG CHANNELS	NO. OF DIGITAL CHANNELS	NO. OF I/O CHANNELS	NO. OF POWER CHANNELS
100. ICS-910					4	0
101. ICS-910					0	0
102. ICS-910					0	0
103. ICS-910					0	0
104. ICS-910					0	0
105. ICS-910					0	0
106. ICS-910					0	0
107. ICS-910					0	0
108. ICS-910					0	0
109. ICS-910					0	0
110. ICS-910					0	0

Figure 26.

We must now choose the Intel iSBC boards which will provide a solution to our system requirements. This is done by referencing the summary of key iSBC configuration parameters to find boards which provide the necessary characteristics. Our first task is to choose a single board computer which meets as many of our needs as is practical, while providing performance characteristics adequate to our needs.

Our first requirement for having support for a single RS232C serial communications channel can be seen to be met by a variety of possible boards. Among the possible boards meeting this requirement are:

iSBC 86/12™ iSBC 80/10A™
 iSBC 80/20™ iSBC 80/20-4™
 iSBC 80/30™

We must look further before a final choice can be made. Again, it can be seen that all candidates also meet the requirement of providing a minimum of one 24-bit wide digital I/O connector. Our decision must be based upon parameters which are not necessarily related to the input or output capabilities. Even though we have not yet developed our software package for our control system, we can safely make some assumptions regarding the completed software package and thus define additional requirements which will enable us to select our desired computer board. The software task will be considerably simplified if we write our programs in a high level language and if we use available drivers for our input and output where they are available. As we will see, the utilization of PL/M and RMX/80™ real-time executive and drivers will make this programming task much less demanding of our time. The trade-off is that these software tools take larger amounts of memory than if we were to write our entire application program in assembly language. Let us make an initial estimate that our system will require about 8K of EPROM and in the neighborhood of 2K of RAM.

Entering this data on the configuration worksheet (Figure 27) enables us to narrow our choice by eliminating the iSBC 80/10A since it does not have sufficient RAM on board.

PROJECT CONFIGURATION WORKSHEET																			
EQUIPMENT PARAMETERS:																			
ITEM	DESCRIPTION	UNIT	QTY	PRICE	TOTAL	QTY	PRICE	TOTAL	QTY	PRICE	TOTAL	QTY	PRICE	TOTAL	QTY	PRICE	TOTAL	QTY	PRICE
10A	iSBC-80/10																		
10B	iSBC-80/20																		
10C	iSBC-80/30																		
10D	CRT																		
10E	2K	1K																	
10F	16K																		
10G	16K																		
10H	16K																		
10I	16K																		

Figure 27.

Since our application is not likely to require extensive math handling capabilities or high speed capabilities, we probably do not need the power found in the iSBC 86/12; so we will remove this product from consideration.

We are now faced with selecting either the iSBC 80/20 board or the 80/30 board for our processor. Each has certain advantages and disadvantages for use in our application. Let's compare these two boards, considering first the iSBC 80/20, then the iSBC 80/30.

iSBC 80/20 board advantages — Slightly lower cost, greater number of I/O lines available.

iSBC 80/30 board advantages — Faster processor, dual ported memory, able to utilize UPI modules.

If the system were to operate in a stand-alone environment and we could be certain that significant expansion would not take place, we would probably choose the iSBC 80/20 computer for our application. If we consider that the system might become a part of a much larger system by future expansions and additions, we should remember that the use of the UPI modules on the iSBC 80/30 computer provides considerable power through multiprocessing capabilities. The dual ported memory can also provide us with the ability to use more sophisticated inter-board communication protocol should the need arise. For the purposes of this application note, we will assume the system is being designed for expansion and we will select the iSBC 80/30 computer.

A good design practice is to provide an extra margin of available memory in the hardware design. Our anticipated RAM memory will use about 2K bytes. The computer will provide us with 4K bytes so we have a considerable margin. This is not true when we look at the amount of EPROM available on the board. Our 8K requirement is identical to

the amount of memory available to us on the board. We should consider the use of an expansion EPROM board or the prospect of having to spend a considerable amount of time reworking our program to get it to fit if we find that we have exceeded our estimates. We will select the option of adding a memory expansion board (it can be deleted if we find that our software requirements are less than estimated).

The computer selection and the memory expansion board data can now be entered onto the configuration worksheet as shown in Figure 28. If needed, the addition of the memory expansion board will allow our EPROM requirements to grow up to 16K bytes.

PROJECT CONFIGURATION WORKSHEET																			
EQUIPMENT PARAMETERS:																			
ITEM	DESCRIPTION	UNIT	QTY	PRICE	TOTAL	QTY	PRICE	TOTAL	QTY	PRICE	TOTAL	QTY	PRICE	TOTAL	QTY	PRICE	TOTAL	QTY	PRICE
10A	iSBC-80/10																		
10B	iSBC-80/20																		
10C	iSBC-80/30																		
10D	CRT																		
10E	2K	1K																	
10F	16K																		
10G	16K																		
10H	16K																		
10I	16K																		
10J	16K																		
10K	16K																		
10L	16K																		
10M	16K																		
10N	16K																		
10O	16K																		
10P	16K																		
10Q	16K																		
10R	16K																		
10S	16K																		
10T	16K																		
10U	16K																		
10V	16K																		
10W	16K																		
10X	16K																		
10Y	16K																		
10Z	16K																		

Figure 28.

The only requirement which we have not met is to assign a board to handle the analog input needs of our temperature sensing circuit. The analog voltage can be calculated and will be found to lie in the neighborhood of 4.6 volts at room temperature. This value will increase toward 5 volts as the temperature of the oven increases. Since we have no requirement for any analog output capabilities, we will choose the Intel® iSBC 711™ Analog Input Board to sense the voltage level. This board can be configured to handle a 5-volt full scale input and will provide a resolution of 12 bits. (If an oven requiring a wide range of temperatures and greater resolution were required, we would have to reconfigure our temperature sensor to provide a wider voltage spread over operating temperatures. For purposes of simplicity and clarity we will assume that our temperature resolution is adequate.)

The configuration worksheet can be filled in to reflect the selection of the analog converter and the total power requirements for the system can be computed as has been done in Figure 29. We now need to select a chassis and power supply in order to complete the application hardware design phase.

The Industrial Chassis

Before the boards can be operated together to form a control system, a means of allowing communica-

SEE INSTRUCTION SHEET

PROJECT CONFIGURATION WORKSHEET

EQUIPMENT PARAMETERS:

YOUR REQUIREMENTS	PRODUCT	MEMORY		SERIAL I/O		PARALLEL INPUT/OUTPUT								ANALOG INPUT/OUTPUT								POWER REQUIREMENTS				COST										
		RAM (bytes)	EPROM (bytes)	TTY	RS232C	IN	OUT	IO	24	DI	DO	LOW	HIGH	VO	IN	OUT	-5V	-12V	5V	12V	LIN	Qty	Qty	Qty												
	1ea. iCS-910															4	0	0	0	0	1	0														
	1ea. iCS-920							0	16																											
	1ea. iCS-930							4	4																											
	CRT					1																														
	TOTAL		2K	5K			1	4	20		1	4									1															
INTEL SOLUTION																																				
	SLOT #1: iSBC 60/30	16K	5K			1	4	20		1	1																									
	SLOT #2: iSBC 416		16K																																	
	SLOT #3: iSBC 711															8					1															
	SLOT #4																																			
	SLOT #5																																			
	SLOT #6																																			
	SLOT #7																																			
	SLOT #8																																			
	SLOT #9																																			
	SLOT #10																																			
	SLOT #11																																			
	SLOT #12																																			
	ROM: EPROM																																			
	I/O DRIVERS																																			
	TERMINATION																																			
	USER SUPP. HARDWARE																																			
	SUBTOTAL																																			

SYSTEM INTEGRATION	SLOTS AVAILABLE	POWER AVAILABLE		
		-5V	-12V	5V
SYSTEM:				
CHASSIS: iCS 10-35	4	4.0	2.0	9.00
CARD CASE				
POWER SUPPLY				
		TOTAL SYSTEM COST		

NOTE:

DI Differential Input
DO Single ended Input
IO Current Output
VO Voltage Output

Figure 29.

tion between the boards and of distributing power among the boards must be found. This requirement is met by specifying a chassis into which the boards will be mounted. The Intel® iCS 80™ Industrial Chassis provides an environment for operating the boards which is specifically designed to operate in an industrial area.

The chassis has been designed to facilitate mounting into either a standard 19-inch RETMA cabinet or it may be rear-panel mounted into an enclosure such as may be found in applications requiring the use of a NEMA electrical enclosure. The card chassis has been mounted in such a manner as to hold the single board computers and expansion modules vertically, facilitating maximum cooling of the boards. Fans are provided to aid the normal convection cooling process. Card racks may be installed into the iCS 80 chassis to expand the card support capability to a maximum of 12 card slots in groups of four. Either an iSBC 635 or 640 power supply can be mounted into the industrial chassis to provide power up to 4 or 12 boards capability, respectively.

Our application design requires the installation of a three board solution, so we will choose the iCS 80 chassis with one iSBC 635™ power supply. We will choose to mount our control system in a standard NEMA 12 enclosure to protect the unit from the industrial environment. We should refer to the *iCS 80 Industrial System Site Planning and Installation Guide* (manual order number 9800798) for complete details for selecting appropriate enclosures and installation instructions.

The +5 volt power needed to support the various termination panels and to supply a reference voltage for the thermistors is available from a barrier strip located on the lower front of the iCS 80 chassis (Figure 30). Our wiring can be routed to this barrier strip for those circuits requiring either 5-volt DC or the system logic common. A fuse holder is provided and a fuse should be installed for system protection. We will install a 2-amp fuse into the holder (our maximum power requirement for external circuitry should be 1.22 amps according to Figure 26).

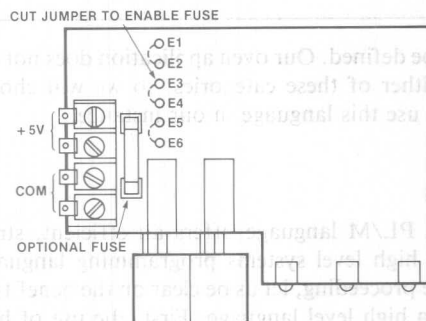


Figure 30. Industrial Chassis DC Power Strip

The remaining terms required in our ladder diagram (Appendix B) consist of a high voltage neutral and a source of switched high voltage power for the heater lamps. Both of these terms are available from the iCS 80 industrial chassis. It is desirable to utilize the same switched power for both the computer system and our external signals, so that we can provide protection to operators when one portion of the system is shut down. A common source will insure that all portions of the system are inactivated if repair is being done. The iCS 80 chassis incorporates a heavy duty industrial key-lock switch for its power switching. The outputs of this switch are available to the user at a terminal barrier strip located on a fold-out panel on the rear of the chassis assembly (refer to Figure 31). We can see that our neutral wire should be connected to terminal 5 (filtered AC low) and the wire for the AC high, wire #10 on the ladder diagram, should be connected to terminal 9. This will provide us with a switched, fused, and filtered power source for our external wiring.

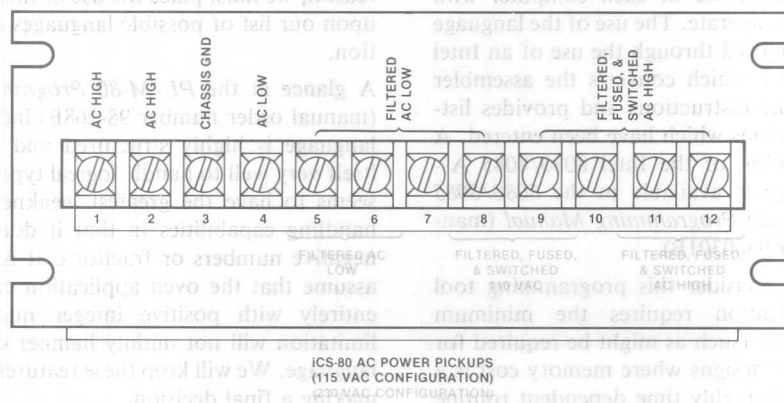


Figure 31. Industrial Chassis AC Power Strip

As we will be installing the chassis into a NEMA enclosure, we will not want to use a standard power cord since this would involve the additional expense of installing a duplex outlet in the cabinet. The power wiring can be installed directly onto the power barrier strip by placing the AC hot wire on barrier number 1, the neutral wire onto barrier number 4, and the ground onto barrier number 3.

The hardware implementation of the system can now be considered to be complete. Before the system can function as a control for the oven temperatures, we must define the relationships between the various pieces of the oven system and we must also define the operator interface with the CRT terminal. Thus, we begin the software phase of our design.

IV. DETERMINATION OF SOFTWARE APPROACH

The task of providing the relationships between the various system components falls into the category of writing the software. Before we actually begin to develop this software, we will define certain guidelines which can be used to organize and simplify the task.

Let us consider the general environment under which our programs will operate. We find that we have essentially two choices in this area. First, we can consider the entire process as a sequential set of predefined operations in which we must perform each operation before moving to the next until finally we complete the sequence and begin again. (This is analogous to using a single stepper switch to design our control system.) Since each oven is independent of the others, we can not afford to use

this approach since we could get tied up waiting for something to happen in a particular oven and would have to ignore the other ovens. The designer familiar with relay design will probably be thinking, at this point, that we should use a separate sequential operation for each oven or device to be controlled. Indeed, this is exactly what we can do with our software by using what is known as a real-time executive. This tool will allocate the computer's resources in such a manner as to provide us with the capability of having independent software programs or tasks operating at what appears to be the same time. We will make our first assumption that our software will be written using such a tool and we will specify that we will operate under Intel's RMX/80 Real-Time Multi-Tasking Executive. We will discuss more detail of this software tool as we develop our programs.

Next, we must consider the language which we will use to actually define our required operation. We have many alternatives from which to choose. Let us look at several of the alternatives in some detail.

Assembler

Assembler language is probably the most basic tool with which we can program a computer. It is considered to be the most efficient user of program memory and processor time. These features are made possible because each assembler instruction line is converted directly into a corresponding machine instruction. From a programming standpoint, assembler language is the most difficult to use since any task must be defined by subdividing that task into a multitude of smaller operations compatible with the available instructions of the computer. To use this language, we must be familiar with the architecture of each computer with which we desire to operate. The use of the language is somewhat simplified through the use of an Intel supplied assembler which converts the assembler code into machine instructions and provides listings of the operations which have been entered. A complete description of the Intel 8080/8085 Assembler Language is available in the *8080/8085 Assembly Language Programming Manual* (manual order number 9800301B).

The user should consider this programming tool when his application requires the minimum amount of memory (such as might be required for very large volume designs where memory cost is a factor) or where a highly time dependent routine

must be defined. Our oven application does not fall into either of these categories, so we will choose not to use this language in our instance.

PL/M

Intel's PL/M language offers an efficient, structured, high level systems programming language. Before proceeding, let us be clear on the benefits of using a high level language. First, the use of high level languages results in reduced development time and cost. High level languages provide the ability to program in a natural algorithmic language. In addition, they eliminate the need to manage register usage or to allocate memory. Second, high level languages provide improved product reliability because programs tend to be written in structured formats and result in a minimum of extraneous branches which might cause testing problems. Finally, their use produces programs which are better documented and are easier to maintain.

On the other hand, high level languages do not optimize the code segments as well as can be done by an experienced assembly language programmer. As a result, most compilers (routines which convert the high level languages into machine executable code) use more program storage than those written by the assembly language programmer. Different languages and compilers require different amounts of memory for the same task.

PL/M-80 is probably one of the most efficient high level languages for use on microcomputers. It has been determined that PL/M-80 users can expect to use between 1.1 to slightly more than 2 times as much program memory as would be used for the same task written in assembly language. For this reason, we must place the use of this language high upon our list of possible languages in this application.

A glance at the *PL/M-80 Programming Manual* (manual order number 98-268B) indicates that the language is highly structured and seems to lend itself very well to handle logical type operations. It seems to have the greatest weakness in its math handling capabilities in that it does not support negative numbers or fractions. It is reasonable to assume that the oven application can be handled entirely with positive integer numbers so this limitation will not unduly hamper our use of this language. We will keep these features in mind when making a final decision.

FORTRAN

Intel's FORTRAN-80 provides the full subset of ANSI FORTRAN 77. In many cases FORTRAN-80 has features that exceed the specifications for both the subset and the full versions of FORTRAN 77. Most of the power of this language lies in its ability to easily handle complex mathematical expressions. Obviously, it does not have any limitations regarding fractions or sign of the numbers involved. It should be used when the application requires the use of mathematical computations. The power of the language, however, means that the use of the language will take a heavy toll of memory allocation. A complete description of the FORTRAN version supported by Intel and its use on the iSBC computers can be found in the *FORTRAN-80 Programming Manual* (order number 9800481A) and in the *ISIS-II FORTRAN-80 Compiler Operator's Manual* (order number 9800480).

It is unlikely that the magnitude of mathematical routines required to control the temperature of our ovens will be complex enough to justify the use of FORTRAN. Keep in mind that, if such a situation were encountered, it is feasible to use a combination of programming languages to create our final module.

BASIC

Certainly the most well known high level programming language today is BASIC. It offers a quick way of applying the computational capabilities of the computer to a wide range of applications. The Intel RMX/80 BASIC-80 is an interpreter designed to operate with Intel's single board computers and contains extended disk handling capabilities. As an interpreter, it differs from other high level languages in that it results in a relatively slower operating solution to an application. It is also not possible to use BASIC to generate multiple independent tasks which can compete for computer resources.

For these reasons, we cannot consider the use of BASIC for a solution to our application.

Final Selection of Language

From the above discussion, it seems clear that our choice for the application being demonstrated is to use PL/M-80 as our programming language.

With this in mind, we can begin the task of actually generating the code which will complete our application and provide an operating control system.

V. DEFINING SOFTWARE TASKS

The software implementation can begin as soon as we have broken our control functions into independent "tasks". We can then handle each task separately as though it were the only thing which had to be done by the control system. In the event that we find that one of our tasks must communicate with or be interlocked with another, we will handle this need through the use of "exchanges". The "exchange" can be thought of as a mailbox into which messages are deposited and picked up by the various tasks. These messages convey the necessary information between the otherwise independent programs. When all tasks have been coded, we will combine them using the facilities of RMX/80.

Our oven application can be broken down into three functional areas or tasks. These are:

1. The Control Task which will be used to actually sense the oven temperature and to provide the required responses to the heaters and the indicator lamps.
2. The CRT Update Task will be used to provide a "snapshot" of the system operations to a person viewing the CRT terminal.
3. The Parameter Update Task will be used to examine and update the oven setpoints and tolerances.

The choice of these three tasks has been essentially arbitrary in nature. Certainly, other choices and groupings of functions could easily have been made. We will use these choices for our example and will proceed with our development accordingly.

We have two other supporting tasks which must be included in our system. Fortunately, these tasks are predefined and fully supported within RMX/80's libraries; thus we need not write these functions. The two supporting tasks are:

4. A Terminal Handler Task to support the actual interface to the CRT terminal. It provides echo of input characters and signals when data is ready to be read. It will output messages to the terminal and signal when all characters requested have been sent.
5. An Analog I/O Driver Task to request and handle the handshaking which is required to communicate with the analog input board. It will signal us when data has been input and is available for use by our user written tasks.

We can proceed with the implementation of each of our three tasks which we have defined. The first step with each will be to develop a flowchart which shows the required operations to implement that task. This flowchart will show any intertask communications or exchanges that may be required with other tasks. The flowchart can then be coded using the facilities provided by our programming language.

Oven Control Task

The sequence of operations required to perform the control task can be defined using the flowchart shown in Figure 32. Let us examine the required steps in more detail.

An arbitrary decision has been made to only sample and control the ovens once each second. This will allow some time for the system to respond once a heater output has been set. The first step in our control task is to wait for one second to elapse.

Our next subtask should be to read the status of the various oven control switches on the operator's control panel. This item could wait until a later time, but there is no harm in handling it at this time.

Next, we see a block indicating the input of data regarding the current oven temperatures. This oven temperature data will certainly be used by the task handling the snapshot display on the CRT so we must give some consideration to the validity of the data. While we are in the process of getting the data and converting it to engineering units (next step), there will be periods during which the stored temperature data does not reflect the actual oven temperature. An example might be when we are actually moving the 16 bits of the temperature since we can only move data 8 bits at a time. During this period, we would not want another task to use the data and since each task is going to operate independent of others, we must provide some type of lockout of the data while we are operating on the temperatures (an alternative would be to have each task get its own temperature from the A/D converter and convert it to engineering units, but this would seem to waste memory and computer time). We can provide this lockout by creating an exchange to communicate with other tasks. If we make a message available in this exchange when the data is valid and cause no messages to be available when the data is nonvalid, we can effectively lock out tasks from using the data when it is in the process of being updated. This is done by requiring

those tasks to test for the presence of a message at the exchange before they get the temperature data. If no message is present, they must wait until one is placed into the exchange before proceeding. Just before we update the temperatures we will fetch the message from the exchange, leaving it empty while we work on the data. later we will again restore the message when the update is complete.

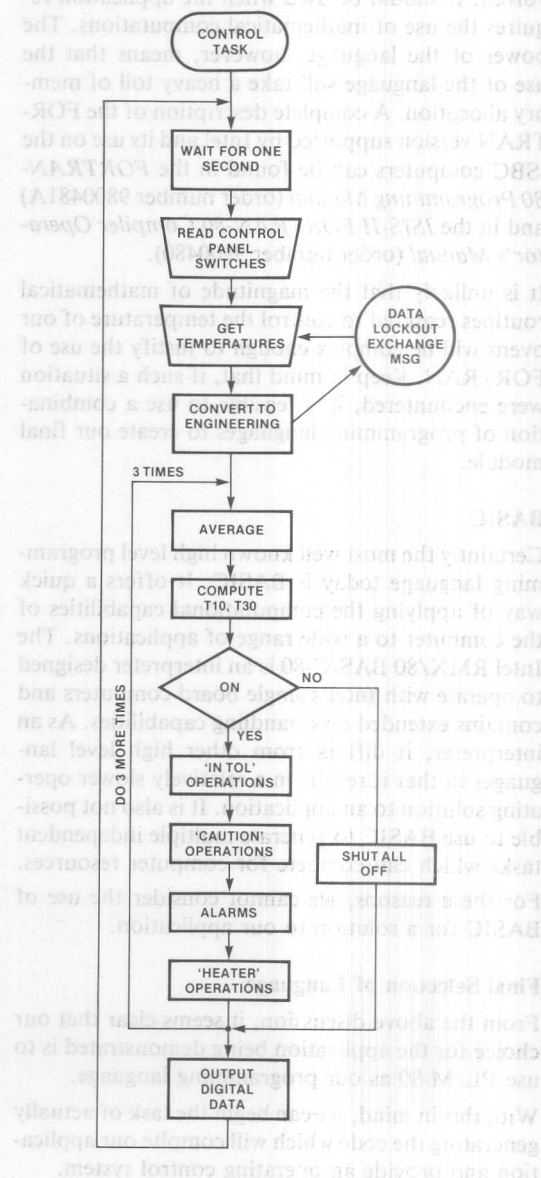


Figure 32. Control Task Flowchart

The number obtained from the analog converter provides us with a value which is proportional to the temperature of the oven. Our next step is to convert this number into engineering units. Unfortunately, the voltage and temperature are not related in a linear fashion since the thermistor is a nonlinear device. We will have to develop a technique to obtain a corrected value. For the purposes of this application note and in an attempt to keep the application as simple as possible, we have chosen to utilize a single table look-up to perform this conversion. Alternatives might have been to utilize FORTRAN routines to mathematically perform the conversion or to have separate tables for each oven. Once the conversion has been made, we must return a message to the data lockout exchange to allow other tasks access to the data.

Because we must deal with four ovens, the operations related to each individual oven must be performed four times, once for each. This is easily handled as we will see, since PL/M is a block structured language. Our flowchart need only remind us that the operations need be done four times.

The next step has been defined as performing some digital filtering of the temperature by averaging the current temperature with the temperature of one second ago. This filtered value will be used to perform subsequent computations and to make future decisions.

We have defined earlier in our definition of the control algorithm that we would use a derivative control. We have chosen to project the temperature ahead for a period of 10 and 30 seconds. We must calculate the rate of change and the temperatures in 10 and 30 seconds so that this data will be available when needed.

Now that the calculations have been made to determine numeric values required for the decision making process, we must begin the process of determining the status of each indicator and oven heater. A test will be made of the oven run switch and if it is found to be turned off, we will turn off all indicators and the oven heater associated with that oven. If the switch is found to be turned on, we will set the status of the "in tolerance", "caution", and "alarm" indicators according to our oven control algorithm. The oven heater will be turned on or off according to the projected temperature in 30 seconds.

Rather than output the individual oven indicator and heater data four times (once for each oven), we

will perform the computations associated with making the decision four times (this saves code since we can use the same program steps with only pointers being exchanged). At the end of this time, a single operation will output the data to all ovens and indicators at the same time. Outputting to a computer port will actually cause the device to turn on or off according to whether the output bit is a one or zero.

We will then return to the beginning of our task to wait until another second elapses before we again perform the indicated functions.

Control Task Source Coding — The coding of our tasks is a straightforward procedure once we have prepared a flowchart. Since we are using PL/M-80 and RMX/80, the coding sequence for a task will be as follows:

1. Define any variables or structures which will be used in the module. This involves providing information defining variables as being either an 8 or 16-bit variable and declaring if that variable is to be a part of the task being coded or is to be found in some other task. If any arrays or structures are to be used, they must also be defined. Finally, if any program locations are to be used, they must be declared.
2. The task must be initialized. That is to say that any assumptions which will be made as to initial data values in subsequent instructions must be initially forced to this initial value.
3. The actual task must be coded to match the operations called out in the flowchart.

We will look at some examples of this coding process using the control task flowchart. The complete listing of this module and all modules actually used to provide the oven control system can be found in Appendix C.

At first glance, it would seem that the listing is extremely complex, but as we will see it is made up of straightforward pieces. The listing is made up of three parts as we have mentioned above when defining the steps required to generate a program. The first part (line numbers 1 through 50) is used to define parameters, variables, and external elements. The general types of elements making up this portion fall into typical categories. The first general category consists of DECLARE statements. Examples of typical lines will help explain their meanings (when actually developing the program, this first section was created piecemeal by

making an entry when it was found that a need for that term existed as the execution code in sections two and three were written).

Examples of the "declare" statement are shown below. For example, on line 11 we find:

```
11 1 Declare (n,k) byte;
```

This means that the variables "n" and "k" are being defined as terms which represent numbers or data which is one byte or 8 bits wide. The "11" is the program line number, and the "1" indicates that we are in the first level of nesting.

We can also see the use of the "literal" expressions such as used in line 4. The expression:

```
4 1 DECLARE FALSE LITERALLY '00H';
```

means that we are creating a new instruction called "false" and that its meaning is to be interpreted by the compiler as being equivalent to the value of zero.

Rather than dwell on the declaration, let us move on to the coding process which was used to generate the actual program. Keep in mind that the use of PL/M-80 requires that all terms used be declared in the program module. Refer to the *PL/M-80 Programming Manual* (order number 9800268B) for a full description of the PL/M language.

Program Initialization — The initialization portion of the program can be found on lines 51 through 59 of the control task program listing. This section is used to initialize data and to provide known entry conditions before we enter the repetitive program loop. This code is only executed when the system is reset or when the power is turned on. The control task requires two types of initializations; one to initialize the computer's output port and the other to set up the A/D converter. The requirements for each can be found in the RMX/80 User's Guide and the *iSBC 80/30 Single Board Computer Hardware Reference Manual* (order number 9800611A). Actual instruction examples are given in these manuals for the initialization operations.

Program Body — The program which actually provides the control operations can be found on lines 60 through 126 of the program listing for the control task. It has been divided into sections which correspond directly to the flowchart that was prepared earlier. Most instructions in PL/M-80 language follow closely the English structure which describes what is being done. The exceptions generally follow definite predefined formats. The for-

mat such as used on line 61 to wait for one second to elapse is an example of one such exception. Any time we desire to wait for a definite time period, we use an instruction of the form:

```
MSGSPTR=RQWAIT (.DUMMYEXCH, TIME DELAY);
```

Whatever time delay we wish to use is expressed in increments of 50 msec time periods. Our example requires a time delay of one second so we will use the delay notation of $1.0/0.050 = 20$ time units (this command is actually calling upon the RMX/80 executive to handle the delay).

The oven enable switch data has been defined by us to be routed by the hardware to the computer port "EA" which converts to a decimal number, 234. If we define an internal memory location for this data and call it BLOCK0, then we can get the oven switch data by using an input statement. Since the data sense is inverted through the hardware, we can provide meaningful internal data if the signal is re-inverted as it is loaded into memory. The instruction on line 62 of the control task listing performs this task.

We are now ready to get the analog data from the A/D converter. Our flowchart shows that we must lock out the other tasks from access to the temperature data during this time period, so we must first remove the enable message from the exchange in which it is stored. Messages are removed from an exchange by using an instruction of the form:

```
STORAGE=RQWAIT (EXCHANGE NAME,0)
```

Line 63 of the program listing means that we will get a message from our storage exchange which is called "Temp\$lockout\$exch" and store it in a memory storage area called "Lockout". Now, no other task can get a message from this exchange since it is empty, so it is permissible to operate on the temperature data. (Note how similar this command is to the one used to wait for a delay. Indeed, this is the same request for RMX/80, but it requests a time delay of zero.)

During the initialization, we built a message defining the characteristics of the analog signals and of the analog conversion board which we are using. Remember that we have indicated that the task of getting this data from the board is provided to us by one of RMX/80's predefined drivers. All that is necessary at this time is to inform that driver of our desire to get data, then wait until it has done its job and the data is available for us. The actual communication between our applications task and the analog driver is done using the idea of an exchange similar to that we have used to lockout the data.

We will send a message to the analog driver telling it what we want it to do, then we will wait until it sends a message back to one of our exchanges telling us that it is done. The format for sending a message to an exchange always follows the form:

CALL RQSEND (EXCHANGE NAME, MESSAGE NAME);

Line 64 of the listing shows that we have requested the input of the analog data since we have sent our message, Convert, to the analog driver's exchange which is called RQAIEX. We will wait until the operation is complete by using the line of code shown on the listing line 65. This is the same operation type that we used to get our message back providing a lockout earlier. The program will wait until a message is available before continuing.

The data must now be converted into engineering units. We earlier indicated that we would use a table lookup to perform the linearization, so we have included this table as a part of our program at line 50. The offset into the table corresponding to our temperature must be determined so that the correct value can be stored. Because we have four ovens, we will perform the operation four times with the data each time corresponding to the appropriate oven. These operations can be followed on lines 77 through 81 of the listing.

Lines 67 through 76 are used to establish an offset to be applied to the analog temperature data when the system is running. This program is only designed to be used during the start-up operations and is activated when a message containing a calibration request and current temperature is sent to its exchange.

The temperature lockout must be removed to enable other tasks to use this data. This is done on line 82 by sending the message back to the exchange used for intertask lockout communications.

The remainder of the program follows the flowchart and the operations can be followed using a flowchart and the listing. Each element of the flowchart corresponds to a block of code on the listing.

CRT Update Task Development

Earlier, we stated that the CRT update task would be used to allow the operator to view a "snapshot" of the four ovens. Let us turn our attention to developing the software which is required to accomplish this. We can begin by defining the elements which we feel should be displayed, then defining the format to actually be used with the CRT terminal.

Obviously, we need to provide the current temperature of each oven on our display screen. If we display the actual temperature, it seems reasonable to assume that we should also show the setpoint so that a determination can be made as to how well the system is performing. The control algorithm has been defined to use an allowable range to determine system outputs, so it would seem wise to also show this parameter. Finally, we should inform the viewer of the status of the oven so that he will realize that the reason an oven temperature is low is because the oven is off rather than an oven malfunction. Other items could be added if desired by the system designer, depending upon the total system requirements or the characteristics of the users.

We can now prepare a drawing of the CRT display to generate a layout of our desired characters and to generate an aesthetic display for viewing during operation. This drawing can be found in Figure 33.

Several techniques are available to output the required displays to the terminal. A decision must be made as to the frequency of screen updates; will we constantly refresh the data or do it only at certain intervals of time? If the terminal has the ability to disable the cursor, it makes sense to update data continuously. If the cursor cannot be disabled, its movement tends to be distracting, so the updates should be kept to a minimum. The terminal used for the application note did not have a disable feature, so we will make the decision to only update the screen once each second.

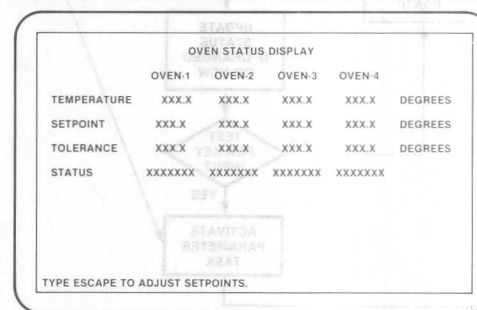


Figure 33. CRT Status Display Layout

The decision to delay updates leads us to make another decision regarding the screen updates. If we only update a line which has data which has changed since the last update, the cursor movements will be kept at a minimum since it is unlikely that all parameters will ever change each second.

A flowchart can now be prepared showing the steps required to implement the CRT update task. This flowchart is shown in Figure 34. The coding of the program to support this task can be found in Appendix C. The development is identical with that which we described in the sections regarding the control task. Again, the software is divided into three parts, the declaration statements from lines 1 to 81, the initialization on lines 82 to 87, and the actual task code on lines 88 to 207.

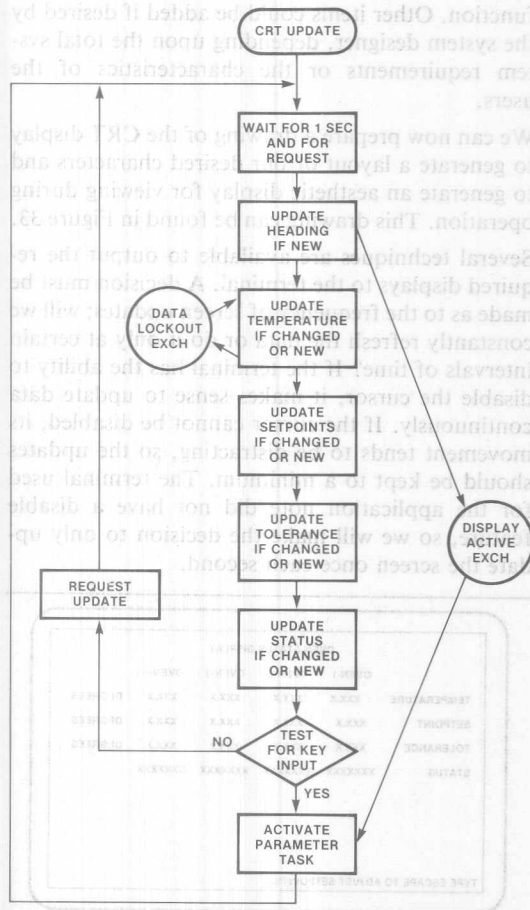


Figure 34. CRT Status Flowchart

A technique to exit from the CRT update mode and to get into a mode which will allow modification of the parameters has been introduced into the program and the display format. This is in the form of a message on the bottom of the screen requesting the entry of an escape character to adjust setpoints. The software has been written in such a

manner as to test for a character input from the keyboard and if one is found corresponding to that character, the update task will allow the parameter update task to take control of the terminal (lines 190 to 204 of the listing).

Parameter Update Task

The parameter update task is used to actually allow the modification of the setpoints and the tolerances associated with each oven. A second use of the task is to provide a tool for establishing the zero offset associated with each analog channel so that an offset into the temperature linearization table can be computed by the control task.

Figure 35 shows the flowchart which describes the steps required to perform these operations. When the task has been completed, we will return to the CRT update task.

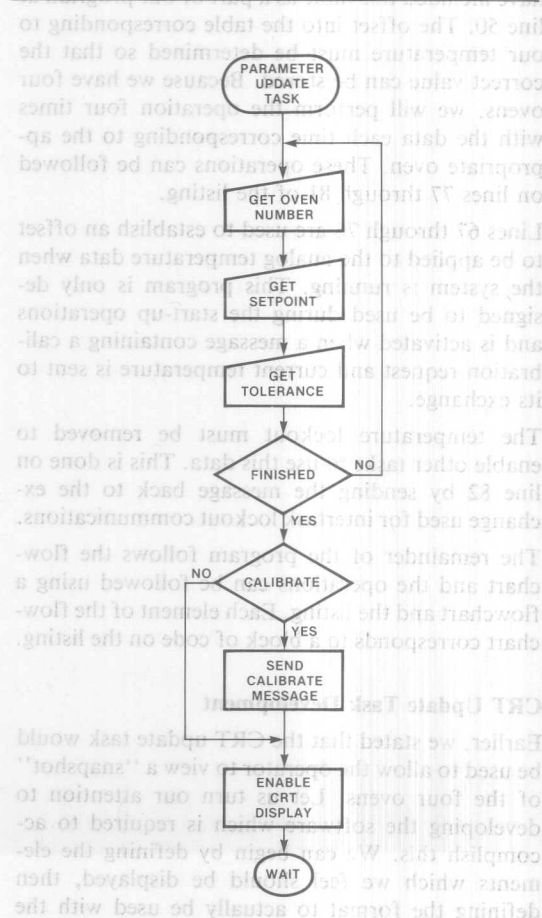


Figure 35. Parameter Update Flowchart

The program code for this task can be found in Appendix C and again follows the formats which we have discussed earlier. No attempt will be made in this document to provide a narrative of the listing since it follows the flowchart in development.

Support Programs

Three subprograms (procedures) have been written which provide functions which are common to the three tasks. This has been done to minimize repeating code segments thus saving as much memory as possible. These three subprograms support:

1. Conversion of a decimal string from the terminal into a binary number. This program is called ASC\$2\$BINARY and can be found in Appendix C.
2. Storage for common variables used by more than one task. These variables could easily have been included in other tasks but a purely arbitrary decision was made to include them in a separate module.
3. Conversion of binary numbers into a decimal string suitable for output to the terminal. This program is called DEC\$REP and is found in Appendix C.

We now have completed the coding of the software to support our oven application. We must finish by combining all the software together to form a single loadable module.

VI. FINAL IMPLEMENTATION

When all code was linked and loaded to form an executable program module, it was found that the system required 9,041 bytes of EPROM and 1,735 bytes of RAM. These values fall within our hardware capabilities and will require that we program and insert nine EPROMs into the EPROM expansion card.

The system can now be tested and installed to control the ovens of our application. The actual system described in this application note has been constructed and tested. It has been found to control the oven temperatures of four ovens and performs as we anticipated when we developed our control strategy earlier in this application note.

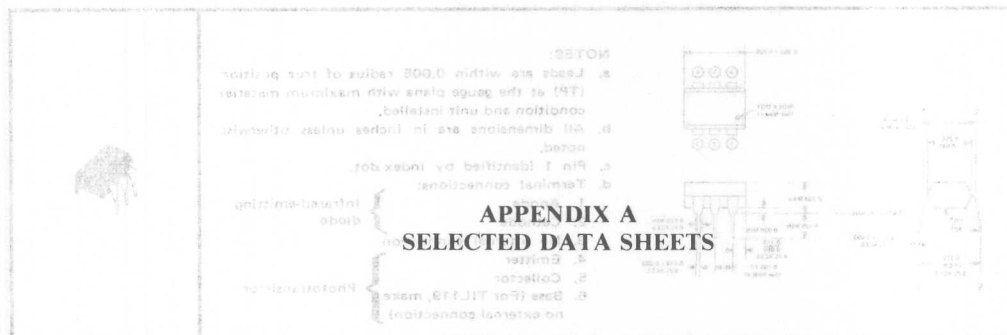
VII. CONCLUSION

We have shown how Intel's single board computers, industrial chassis, termination panels, and software can be configured to provide a solution to a typical control application. We have seen how the development of a solution to a control problem can proceed along a predetermined and logical path. Truly, the utilization of the microprocessors can lead to optimum and cost effective solutions to control applications.

- Gallium Arsenide Diode Infrared Source Optically Coupled to a Silicon N-P-N Darlington-Connected Phototransistor
- High Direct-Current Transfer Ratio . . . 300% Minimum at 10 mA
- Base Lead Provided for Conventional Transistor Biasing
- High-Voltage Electrical Isolation . . . 1500-Volt Rating
- Plastic Dual-In-Line Package
- Typical Applications Include Remote Terminal Isolation, SCR and Triac Triggers, Mechanical Relays, and Pulse Transformers

mechanical data

The package consists of a gallium arsenide infrared-emitting diode and an n-p-n silicon darlington-connected phototransistor mounted on a 6-lead frame encapsulated within an electrically nonconductive plastic compound. The case with wirebond soldering temperature with no deformation and device performance characteristics remain stable when operated in high humidity conditions. Unit weight is approximately 0.85 grams.



APPENDIX A SELECTED DATA SHEETS

absolute maximum ratings at 25°C free-air temperature (unless otherwise noted)

1.5 kV	Infrared Output Voltage
30 V	Collector-Bias Voltage (TIL113)
30 V	Collector-Emitter Voltage (See Note 1)
3 V	Emitter-Collector Voltage
3 V	Emitter-Bias Voltage (TIL113)
3 V	Input-Output Reverse Voltage
100 mA	Input-Output Continuous Forward Current at (or below) 25°C Free-Air Temperature (See Note 2)
	Continuous Power Dissipation at (or below) 25°C Free-Air Temperature:
150 mW	Infrared-Emitting Diode (See Note 3)
150 mW	Phototransistor (See Note 4)
250 mW	Total Infrared-Emitting Diode plus Phototransistor (See Note 5)
55°C to 150°C	Storage Temperature Range
250°C	Lead Temperature 1/16 inch from Case for 10 Seconds

- NOTES: 1. This value applies when the base-emitter diode is open-circuited.
2. Derate linearly to 100°C free-air temperature at the rate of 1.33 mW/°C.
3. Derate linearly to 100°C free-air temperature at the rate of 2 mW/°C.
4. Derate linearly to 100°C free-air temperature at the rate of 2 mW/°C.
5. Derate linearly to 100°C free-air temperature at the rate of 0.33 mW/°C.

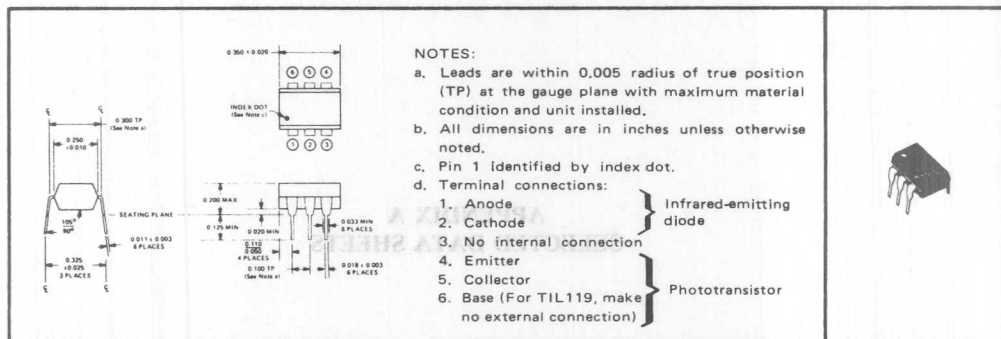
TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5015 • DALLAS, TEXAS 75205

Reprinted with permission from Texas Instruments, March, 1978. All rights reserved.

- Gallium Arsenide Diode Infrared Source Optically Coupled to a Silicon N-P-N Darlington-Connected Phototransistor
- High Direct-Current Transfer Ratio . . . 300% Minimum at 10 mA
- Base Lead Provided for Conventional Transistor Biasing
- High-Voltage Electrical Isolation . . . 1500-Volt Rating
- Plastic Dual-In-Line Package
- Typical Applications Include Remote Terminal Isolation, SCR and Triac Triggers, Mechanical Relays, and Pulse Transformers

mechanical data

The package consists of a gallium arsenide infrared-emitting diode and an n-p-n silicon darlington-connected phototransistor mounted on a 6-lead frame encapsulated within an electrically nonconductive plastic compound. The case will withstand soldering temperature with no deformation and device performance characteristics remain stable when operated in high humidity conditions. Unit weight is approximately 0.52 grams.



absolute maximum ratings at 25°C free-air temperature (unless otherwise noted)

Input-to-Output Voltage	±1.5 kV
Collector-Base Voltage (TIL113)	30 V
Collector-Emitter Voltage (See Note 1)	30 V
Emitter-Collector Voltage	7 V
Emitter-Base Voltage (TIL113)	7 V
Input-Diode Reverse Voltage	3 V
Input-Diode Continuous Forward Current at (or below) 25°C Free-Air Temperature (See Note 2)	100 mA
Continuous Power Dissipation at (or below) 25°C Free-Air Temperature:	
Infrared-Emitting Diode (See Note 3)	150 mW
Phototransistor (See Note 4)	150 mW
Total (Infrared-Emitting Diode plus Phototransistor, See Note 5)	250 mW
Storage Temperature Range	-55°C to 150°C
Lead Temperature 1/16 Inch from Case for 10 Seconds	260°C

- NOTES:
- This value applies when the base-emitter diode is open-circuited.
 - Derate linearly to 100°C free-air temperature at the rate of 1.33 mA/°C.
 - Derate linearly to 100°C free-air temperature at the rate of 2 mW/°C.
 - Derate linearly to 100°C free-air temperature at the rate of 2 mW/°C.
 - Derate linearly to 100°C free-air temperature at the rate of 3.33 mW/°C.

TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5012 • DALLAS, TEXAS 75222

Reprinted with permission from Texas Instruments, March, 1979. All rights reserved.

TYPES TIL113, TIL119

OPTO-COUPLEDERS

electrical characteristics at 25°C free-air temperature

PARAMETER	TEST CONDITIONS†	TIL113			TIL119			UNIT
		MIN	TYP	MAX	MIN	TYP	MAX	
$V_{(BR)CBO}$ Collector-Base Breakdown Voltage	$I_C = 10 \mu A$, $I_E = 0$, $I_F = 0$	30						V
$V_{(BR)CEO}$ Collector-Emitter Breakdown Voltage	$I_C = 1 mA$, $I_B = 0$, $I_F = 0$	30			30			V
$V_{(BR)EBO}$ Emitter-Base Breakdown Voltage	$I_E = 10 \mu A$, $I_C = 0$, $I_F = 0$	7						V
$V_{(BR)ECO}$ Emitter-Collector Breakdown Voltage	$I_E = 10 \mu A$, $I_F = 0$				7			V
$I_{C(on)}$ On-State Collector Current	$V_{CE} = 1 V$, $I_B = 0$, $I_F = 10 mA$	30	100					mA
$I_{C(off)}$ Off-State Collector Current	$V_{CE} = 2 V$, $I_F = 10 mA$				30	160		nA
$I_{C(off)}$ Collector Current	$V_{CE} = 10 V$, $I_B = 0$, $I_F = 0$			100			100	nA
h_{FE} Transistor Static Forward Current Transfer Ratio	$V_{CE} = 1 V$, $I_C = 10 mA$, $I_F = 0$		15,000					
V_F Input Diode Static Forward Voltage	$I_F = 10 mA$			1.5			1.5	V
$V_{CE(sat)}$ Collector-Emitter Saturation Voltage	$I_C = 125 mA$, $I_B = 0$, $I_F = 50 mA$			1				V
$V_{CE(sat)}$ Saturation Voltage	$I_C = 10 mA$, $I_F = 10 mA$						1	V
r_{IO} Input-to-Output Internal Resistance	$V_{in-out} = \pm 1.5 kV$, See Note 6	10^{11}			10^{11}			Ω
C_{IO} Input-to-Output Capacitance	$V_{in-out} = 0$, $f = 1 MHz$, See Note 6	1	1.3		1	1.3		pF

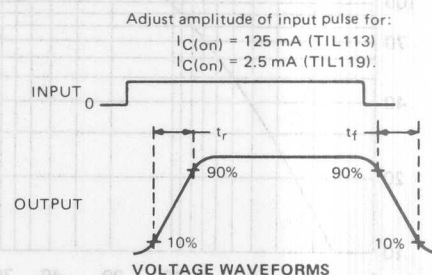
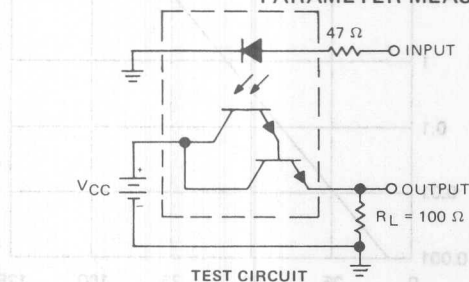
NOTE 6: These parameters are measured between both input-diode leads shorted together and all the phototransistor leads shorted together.

†References to the base are not applicable to the TIL119.

switching characteristics at 25°C free-air temperature

PARAMETER	TEST CONDITIONS	TIL113			TIL119			UNIT
		MIN	TYP	MAX	MIN	TYP	MAX	
t_r Rise Time	$V_{CC} = 15 V$, $I_{C(on)} = 125 mA$		50					μs
t_f Fall Time	$R_L = 100 \Omega$, See Figure 1		50					μs
t_r Rise Time	$V_{CC} = 10 V$, $I_{C(on)} = 2.5 mA$				50			μs
t_f Fall Time	$R_L = 100 \Omega$, See Figure 1				50			μs

PARAMETER MEASUREMENT INFORMATION



NOTES: a. The input waveform is supplied by a generator with the following characteristics: $Z_{out} = 50 \Omega$, $t_r \leq 15 ns$, duty cycle $\approx 1\%$, $t_w = 100 \mu s$.

b. The output waveform is monitored on an oscilloscope with the following characteristics: $t_r \leq 12 ns$, $R_{in} \geq 1 M\Omega$, $C_{in} \leq 20 pF$.

FIGURE 1—SWITCHING TIMES

TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5012 • DALLAS, TEXAS 75222

TYPES TIL113, TIL119 OPTO-COUPLED

TYPICAL CHARACTERISTICS

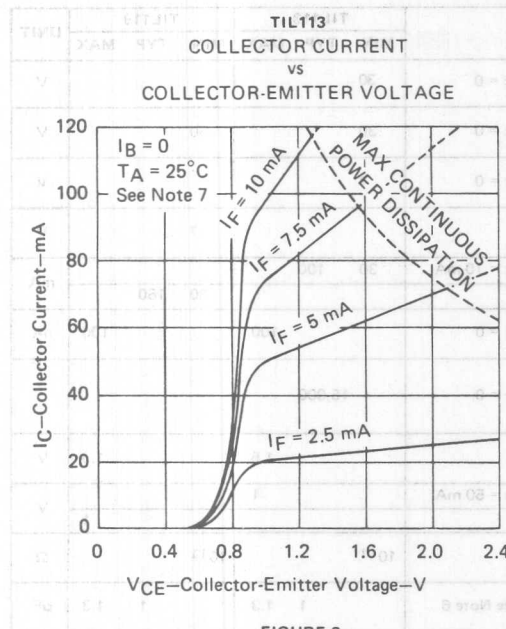


FIGURE 2

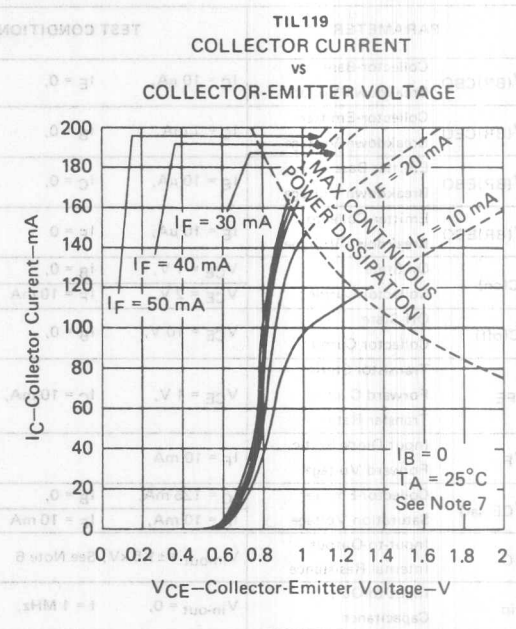


FIGURE 3

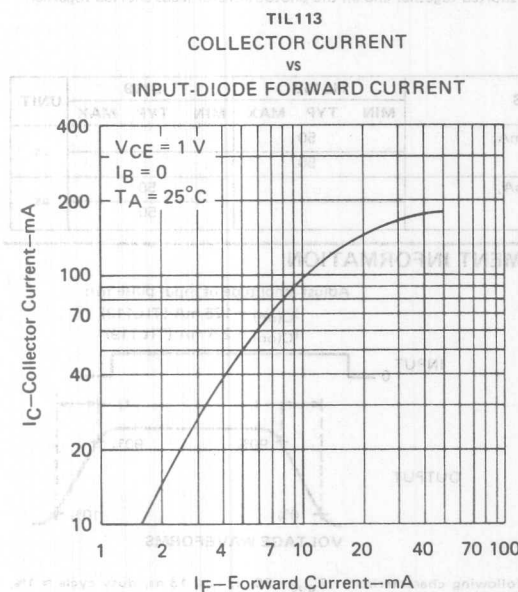


FIGURE 4

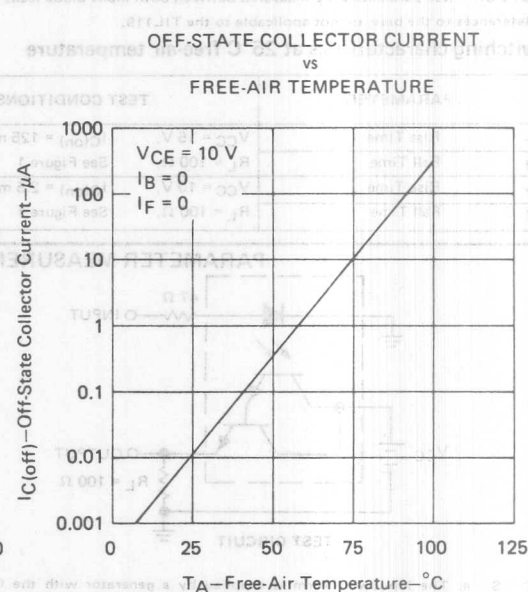


FIGURE 5

NOTE 7: Pulse operation of input diode is required for operation beyond limits shown by dotted line.

TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5012 • DALLAS, TEXAS 75222

TYPES TIL113, TIL119

OPTO-COUPLEDERS

TYPICAL CHARACTERISTICS

TIL113
RELATIVE COLLECTOR-EMITTER
SATURATION VOLTAGE
vs
FREE-AIR TEMPERATURE

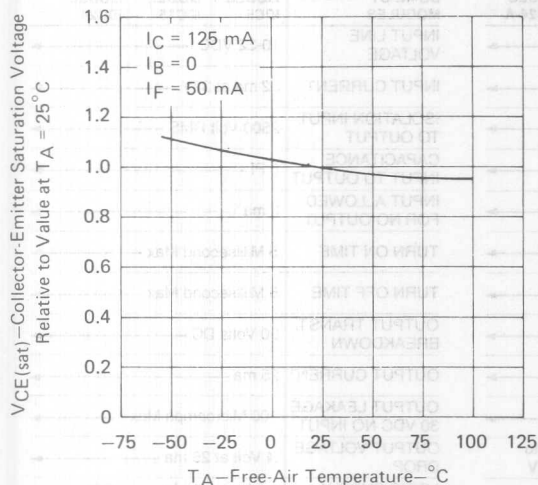


FIGURE 6

TIL113
TRANSISTOR STATIC FORWARD
CURRENT TRANSFER RATIO
vs
COLLECTOR CURRENT

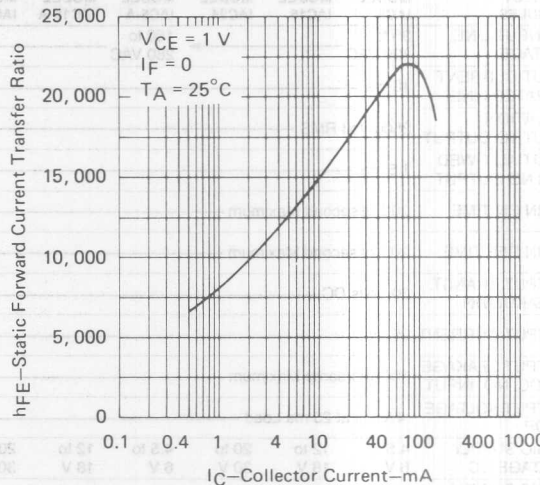


FIGURE 7

INPUT DIODE FORWARD
CONDUCTION CHARACTERISTICS

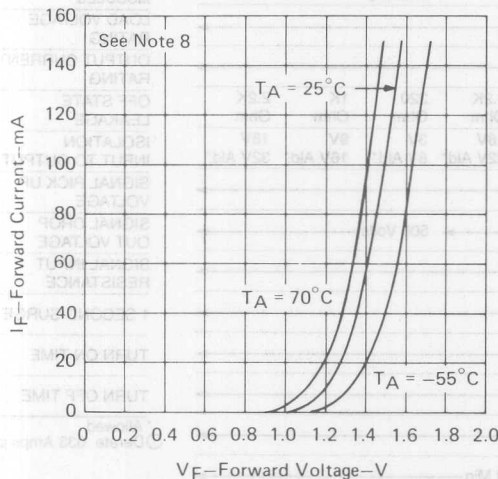


FIGURE 8

NOTE 8: This parameter was measured using pulse techniques. $t_w = 1$ ms, duty cycle $\leq 2\%$.

TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5012 • DALLAS, TEXAS 75222

PRINTED IN U.S.A.

Ti cannot assume any responsibility for any circuits shown or represent that they are free from patent infringement. TEXAS INSTRUMENTS RESERVES THE RIGHT TO MAKE CHANGES AT ANY 1 IN ORDER TO IMPROVE DESIGN AND TO SUPPLY THE BEST PRODUCT POSSI

I/O Module Detail

Electrical Specifications

AC INPUT MODULES	MODEL IAC5	MODEL IAC15	MODEL IAC24	MODEL IAC5-A	MODEL IAC15-A	MODEL IAC24-A
AC INPUT LINE VOLTAGE	95 to 130 VAC			180 to 280 VAC		
INPUT CURRENT AT RATED LINE	10 ma					
ISOLATION INPUT TO OUTPUT	2500 Volt RMS					
INPUT ALLOWED FOR NO OUTPUT	1.5 ma					
TURN ON TIME	20 Millisecond Maximum					
TURN OFF TIME	20 Millisecond Maximum					
OUTPUT TRANST. BREAKDOWN	30 Volts DC					
OUTPUT CURRENT	25 ma					
OUTPUT LEAKAGE 30VDC, NO. INPUT	100 Microamp Maximum					
OUTPUT VOLTAGE DROP	.4 Volts at 25 ma Load					
LOGIC SUPPLY VOLTAGE DC	4.5 to 6 V	12 to 18 V	20 to 30 V	4.5 to 6 V	12 to 18 V	20 to 30 V
LOGIC SUPPLY CURRENT	12 ma	15 ma	18 ma	12 ma	15 ma	18 ma

AC OUTPUT MODULES	MODEL OAC5	MODEL OAC15	MODEL OAC24	MODEL OAC5-A	MODEL OAC15-A	MODEL OAC24-A
LINE VOLTAGE	12 to 140 VAC			24 to 280 VAC		
CURRENT RATING	3 Amps ^①					
1-CYCLE SURGE	55 Amps Peak					
SIGNAL INPUT RESISTANCE	220 Ohm	1K Ohm	2.2K Ohm	220 Ohm	1K Ohm	2.2K Ohm
SIGNAL PICKUP VOLTS DC	3V 8V Ald.*	9V 16V Ald.*	18V 32V Ald.*	3V 8V Ald.*	9V 16V Ald.*	18V 32V Ald.*
SIGNAL DROPOUT VOLTS DC	1 Volt					
PEAK REPETITIVE VOLTAGE	400V			500 Volts		
MAXIMUM CONTACT DROP	1.6V					
OFF STATE LEAKAGE	5 ma RMS					
MINIMUM LOAD CURRENT	20 ma					
ISOLATION INPUT TO OUTPUT	2500 Volts RMS					
CAPACITANCE INPUT TO OUTPUT	8 Pf					
STATIC DV/DT	200 Volts/Microsecond Min					
COMMUTATING DV/DT	Built in snubber (will commute .5 power factor loads)					

*Allowed

DC INPUT MODULES	MODEL IDC5	MODEL IDC15	MODEL IDC24
INPUT LINE VOLTAGE	10-32 VDC		
INPUT CURRENT	32 ma at 32V		
ISOLATION INPUT TO OUTPUT	2500 Volt RMS		
CAPACITANCE INPUT TO OUTPUT	8 Pf		
INPUT ALLOWED FOR NO OUTPUT	2 ma		
TURN ON TIME	5 Millisecond Max		
TURN OFF TIME	5 Millisecond Max		
OUTPUT TRANST. BREAKDOWN	30 Volts DC		
OUTPUT CURRENT	25 ma		
OUTPUT LEAKAGE 30 VDC NO INPUT	100 Microamps Max		
OUTPUT VOLTAGE DROP	.4 Volt at 25 ma		
LOGIC SUPPLY VOLTAGE	4.5 to 6V	12 to 18V	20 to 30V
LOGIC SUPPLY CURRENT	12 ma	15 ma	18 ma

DC OUTPUT MODULES	MODEL ODC5	MODEL ODC15	MODEL ODC24
LOAD VOLTAGE RATING	60V DC		
OUTPUT CURRENT RATING	3 Amps ^①		
OFF STATE LEAKAGE	1 ma Max		
ISOLATION INPUT TO OUTPUT	2500 V RMS		
SIGNAL PICK UP VOLTAGE	3V 8V Ald*	9V 18V Ald*	18V 28V Ald*
SIGNAL DROP OUT VOLTAGE	1 Volt		
SIGNAL INPUT RESISTANCE	220 Ohm	1K Ohm	2.2K Ohm
1 SECOND SURGE	5 Amps		
TURN ON TIME	500 Microsecond		
TURN OFF TIME	2.5 Millisecond		

* Allowed

① Derate .033 Amps per degree C from 20° C

opto 22

5842 Research Drive, Huntington Beach, California 92649 (714) 892-3313

Reprinted with permission from OPTO 22, March, 1979. All rights reserved.

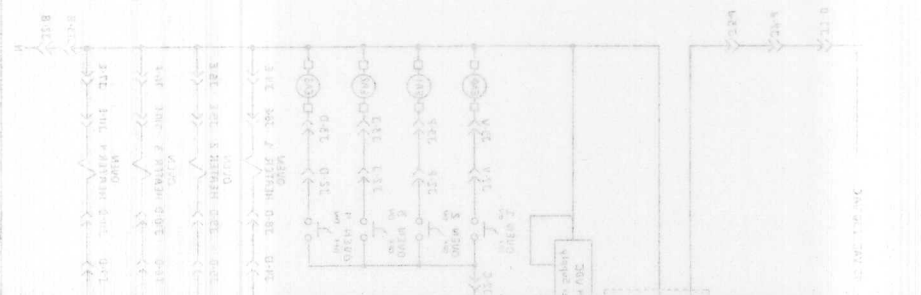
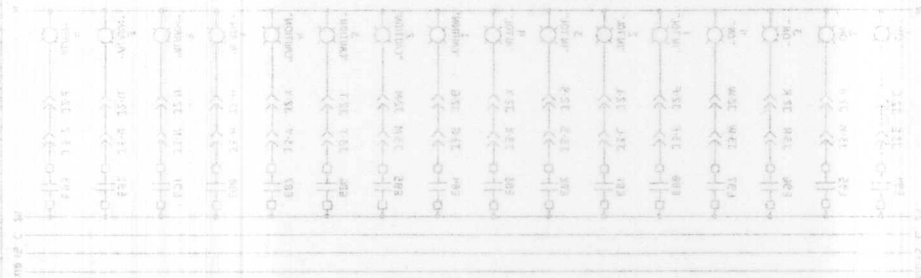
High Voltage DC Output Modules

DC OUTPUT MODULES	MODEL ODC5-A	MODEL ODC15-A	MODEL ODC24-A
LOAD VOLTAGE RATING	200V DC		
OUTPUT CURRENT RATING	1 Amps	30	
OFF STATE LEAKAGE	2 ma Max		
ISOLATION INPUT TO OUTPUT	2500 V RMS		
SIGNAL PICK UP VOLTAGE	3V 8V Ald.*	9V 18V Ald.*	18V 28V Ald.*
SIGNAL DROP OUT VOLTAGE	1Volt		
SIGNAL INPUT RESISTANCE	220 Ohm	1K Ohm	2.2K Ohm
1 SECOND SURGE	5 Amps		
TURN ON TIME	500 Microsecond		
TURN OFF TIME	2.5 Millisecond		

*Allowed

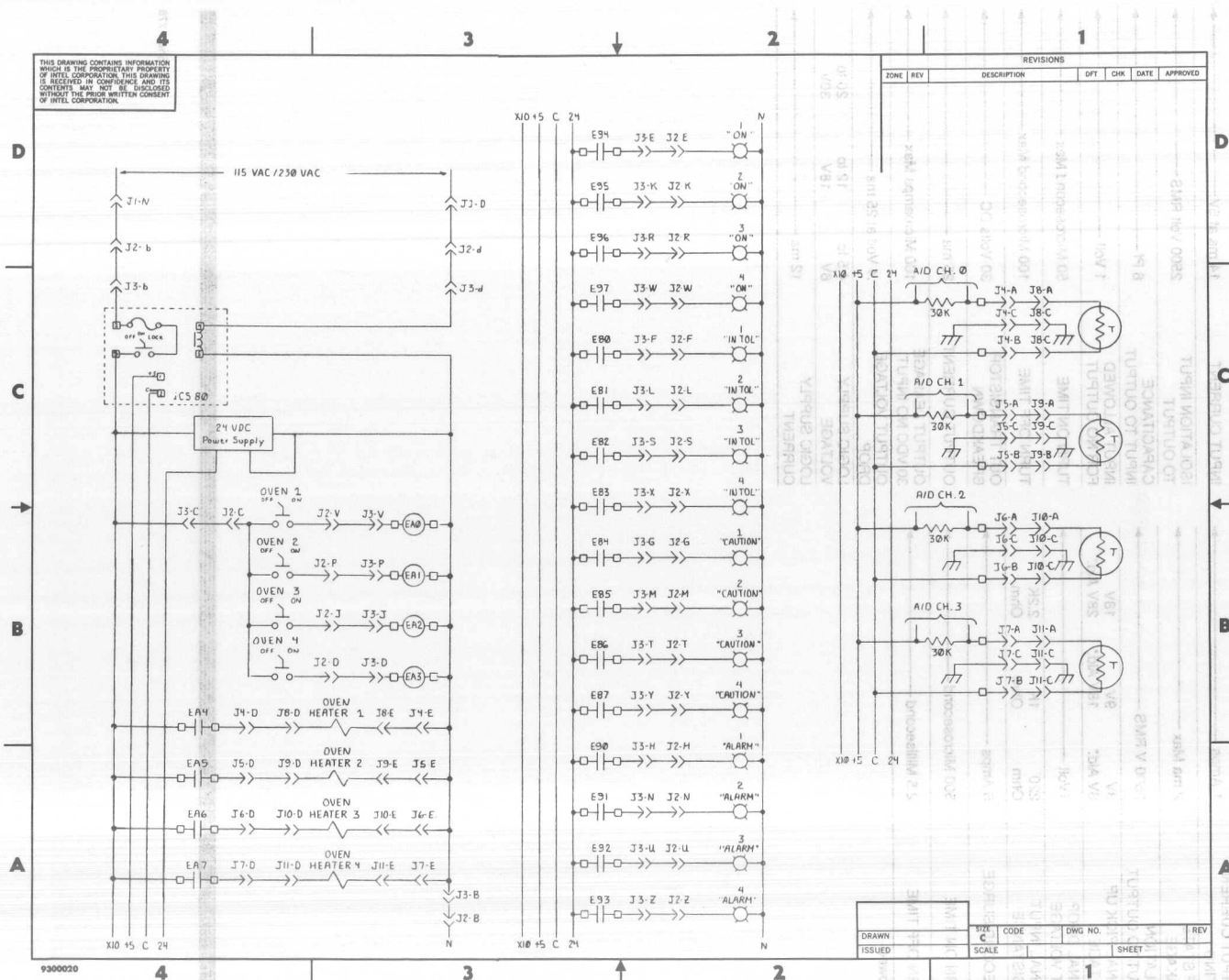
Fast Switching DC Input Modules

DC INPUT MODULES	MODEL IDC5-B	MODEL IDC15-B	MODEL IDC24-B
INPUT LINE VOLTAGE	4-16 VDC		
INPUT CURRENT	14 ma at 5V		
ISOLATION INPUT TO OUTPUT	2500 Volt RMS		
CAPACITANCE INPUT TO OUTPUT	8 Pf		
INPUT ALLOWED FOR NO OUTPUT	1 Volt		
TURN ON TIME	50 Microsecond Max		
TURN OFF TIME	100 Microsecond Max		
OUT TRANSISTOR BREAKDOWN	30 Volts DC		
OUTPUT CURRENT	25 ma		
OUTPUT LEAKAGE 30 VDC NO INPUT	100 Microamps Max		
OUTPUT VOLTAGE DROP	.4 Volt at 25 ma		
LOGIC SUPPLY VOLTAGE	4.5 to 6V	12 to 18V	20 to 30V
LOGIC SUPPLY CURRENT	12 ma		



Data Sheet 778

APPENDIX B LADDER DIAGRAM OF SYSTEM



APPENDIX C **PROGRAM SOURCE LISTINGS**

```

USING INTEL'S INDUSTRIAL CONTROL SERIES IN CONTROL APPLICATIONS

$TITLE('CONTROL TASK')
/*****
* This task handles the control and monitoring of
* four oven chambers.
*****/

1  CONTROLTASK$MODULE:
2  Do;
3  1  DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
4  1  MESSAGE$HEAD ADDRESS,
5  1  MESSAGE$TAIL ADDRESS,
6  1  TASK$HEAD ADDRESS,
7  1  TASK$TAIL ADDRESS,
8  1  EXCHANGE$LINK ADDRESS)';
9  1  DECLARE TRUE LITERALLY 'OFFH';
10 1  DECLARE FALSE LITERALLY 'OFFH';
11 1  DECLARE BOOLEAN LITERALLY 'BYTE';
12 1  DECLARE FOREVER LITERALLY 'WHILE 1';
13 1  DECLARE MSG$HDR LITERALLY 'LINK ADDRESS,
14 1  LENGTH ADDRESS,
15 1  TYPE BYTE,
16 1  HOME$EX ADDRESS,
17 1  RESP$EX ADDRESS';
18 1  DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
19 1  MSG$HDR,
20 1  REMAINDER(1) BYTE)';
21 1  /* AIMSG.ELT - ANALOG INPUT REQUEST MESSAGE FORMAT */
22 1  DECLARE AIMSG LITERALLY 'STRUCTURE(
23 1  MSG$HDR,
24 1  STATUS ADDRESS,
25 1  BASE$PTR ADDRESS,
26 1  CHANNEL$GAIN ADDRESS,
27 1  ARRAY$PTR ADDRESS,
28 1  COUNT ADDRESS,
29 1  ACTUALSCOUNT ADDRESS)';
30 1  /* AITYP.ELT - ANALOG INPUT MESSAGE TYPES */
31 1  DECLARE AIREP LITERALLY '30';
32 1  AISQS LITERALLY '31';
33 1  AISQV LITERALLY '32';
34 1  AITRN LITERALLY '33';
35 1  Declare (n,k) byte;
36 1  Declare (MSG$PTR,LOCKOUT) address;
37 1  Declare (BLOCK0,BLOCK1,BLOCK2,BLOCK3) byte external;
38 1  Declare TOLERANCE(4) address external;
39 1  Declare TEMP(4) address external;
40 1  Declare SETPOINT(4) address external;
41 1  Declare T$AVERAGE(4) address;
42 1  Declare T$LAST(4) address;
43 1  Declare T$LAST$AVERAGE(4) address;
44 1  Declare T$t5(4) address;
45 1  Declare T$tl0(4) address;
46 1  Declare STATUS(4) byte external;
47 1  Declare CRT$DISPLAY$LOCK(5) address external;

```

PROGRAM SOURCE LISTINGS

```

24 1 Declare TEMP$CALIBRATE(5) address external;
25 1 Declare DUMMY$EXCH(5) address external;
26 1 Declare TEMP$LOCKOUT$EXCH(5) address external;
27 1 Declare RQAIEX(5) address external;
28 1 Declare ASD$EXCH(5) address external;
29 1 Declare CONSTANT$LOCKOUT$EXCH(5) address external;
30 1 Declare CRT$STATUS$EXCH(5) address external;
31 1 Declare ALARM$MSG structure (MSG$HDR);
32 1 Declare CONVERT $i$msq;
/* This term is used to convey initial temperatures */
33 1 Declare CAL$TEMP based MSG$PTR structure (
    MSG$HDR,
    CAL address );
34 1 RQWAIT:
    Procedure (EXCH,MESSAGE) address external;
    Declare (EXCH,MESSAGE) address;
35 2
36 2 end RQWAIT;
37 1 RQSEND:
    Procedure (EXCH,MESSAGE) external;
    Declare (EXCH,MESSAGE) address;
38 2
39 2 end RQSEND;
40 1 RQACPT:
    Procedure (EXCH) address external;
    Declare EXCH address;
41 2
42 2 end RQACPT;
43 1 Declare OVEN$IN$TOL(4) byte data (
    01H,02H,04H,08H );
44 1 Declare OVEN$CAUTION(4) byte data (
    10H,20H,40H,80H );
45 1 Declare OVEN$DANGER(4) byte data (
    01H,02H,04H,08H );
46 1 Declare OVEN$ON$MASK(4) byte data (
    01H,02H,04H,08H );
47 1 Declare OVEN$HEATER(4) byte data (
    10H,20H,40H,80H );
48 1 Declare OVEN$RUN(4) byte data (
    10H,20H,40H,80H );
49 1 Declare OFFSET(4) address;
50 1 Declare TABLE(256) address data (
    200,201,202,203,204,205,206,207,208,209,
    210,211,212,213,214,215,216,217,218,
    219,220,221,222,223,224,225,226,227,228,
    229,230,231,232,233,234,235,236,237,238,239,
    240,241,242,243,244,245,246,247,248,249,250,251,
    252,253,254,255,256,257,258,259,260,261,262,263,264,
    265,266,267,268,269,270,271,272,273,274,275,276,277,
    278,279,280,281,282,283,284,285,286,287,288,289,290,291,
    292,293,294,295,296,297,298,299,300,301,302,303,304,305,306,
    307,308,309,310,311,312,313,314,315,316,317,318,319,320,321,322,323,
    324,325,326,327,328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,343,
    344,345,346,347,348,349,350,351,352,353,354,355,356,357,358,359,360,361,362,363,364,
    365,366,367,368,369,370,371,372,373,374,375,376,377,378,379,380,381,382,383,384,
    385,386,387,388,389,390,391,392,393,394,395,396,397,398,399,400,401,402,403,404,405,406,407,408,409,
    410,411,412,413,414,415,416,417,418,419,420,421,422,423,424,425,426,427,428,429,430,431,432,433,434,435,
    436,437,438,439,440,441,442,443,444,445,446,447,448,449,450,451,452,453,454,455,456,457,458,459,460,461,462,463,464,465,466,467,468,469,470,471,472,473,474,475,476,477,478,479,480,481,482,483,484,485,486,487,488,489,490,491,492,493,494,495,496,497,498,499,500,501,502,503,504,505,506,507,508,509,510,511,512,513,514,515,516,517,518,519,520,521,522,523,524,525,526,527,528,529,530,531,532,533,534,535,536,537,538,539,540,541,542,543,544,545,546,547,548,549,550,551,552,553,554,555,556,557,558,559,560,561,562,563,564,565,566,567,568,569,570,571,572,573,574,575,576,577,578,579,580,581,582,583,584,585,586,587,588,589,590,591,592,593,594,595,596,597,598,599,600,601,602,603,604,605,606,607,608,609,610,611,612,613,614,615,616,617,618,619,620,621,622,623,624,625,626,627,628,629,630,631,632,633,634,635,636,637,638,639,640,641,642,643,644,645,646,647,648,649,650,651,652,653,654,655,656,657,658,659,660,661,662,663,664,665,666,667,668,669,670,671,672,673,674,675,676,677,678,679,680,681,682,683,684,685,686,687,688,689,690,691,692,693,694,695,696,697,698,699,700,701,702,703,704,705,706,707,708,709,710,711,712,713,714,715,716,717,718,719,720,721,722,723,724,725,726,727,728,729,730,731,732,733,734,735,736,737,738,739,740,741,742,743,744,745,746,747,748,749,750,751,752,753,754,755,756,757,758,759,760,761,762,763,764,765,766,767,768,769,770,771,772,773,774,775,776,777,778,779,780,781,782,783,784,785,786,787,788,789,790,791,792,793,794,795,796,797,798,799,800,801,802,803,804,805,806,807,808,809,810,811,812,813,814,815,816,817,818,819,820,821,822,823,824,825,826,827,828,829,830,831,832,833,834,835,836,837,838,839,840,841,842,843,844,845,846,847,848,849,850,851,852,853,854,855,856,857,858,859,860,861,862,863,864,865,866,867,868,869,870,871,872,873,874,875,876,877,878,879,880,881,882,883,884,885,886,887,888,889,890,891,892,893,894,895,896,897,898,899,900,901,902,903,904,905,906,907,908,909,910,911,912,913,914,915,916,917,918,919,920,921,922,923,924,925,926,927,928,929,930,931,932,933,934,935,936,937,938,939,940,941,942,943,944,945,946,947,948,949,950,951,952,953,954,955,956,957,958,959,960,961,962,963,964,965,966,967,968,969,970,971,972,973,974,975,976,977,978,979,980,981,982,983,984,985,986,987,988,989,990,991,992,993,994,995,996,997,998,999,1000);

```



```

82 3      Do n=0 to 3;
84 4          TSAVERAGE(n)=(T$LAST(n)+TEMP(n))/2;
/* Project temperatures into the future */
85 4          If TSAVERAGE(n)>=T$LAST$AVERAGE(n)
              then do;
87 5              T$t5(n)=((TSAVERAGE(n)-T$LAST$AVERAGE(n))*5)
                  +T$LAST$AVERAGE(n);
88 5              T$t10(n)=((TSAVERAGE(n)-T$LAST$AVERAGE(n))*10)
                  +T$LAST$AVERAGE(n);
89 5          end;
90 4          else do;
91 5              T$t5(n)=T$LAST$AVERAGE(n)-((T$LAST$AVERAGE(n)
                  -TSAVERAGE(n))*5);
92 5              T$t10(n)=T$LAST$AVERAGE(n)-((T$LAST$AVERAGE(n)
                  -TSAVERAGE(n))*10);
93 5          end;
/* Update stored data */
94 4          T$LAST$AVERAGE(n)=TSAVERAGE(n);
95 4          T$LAST(n)=TEMP(n);
/* Test for active oven */
96 4          MSG$PTR=RQWAIT (.CONSTANT$LOCKOUT$EXCH,0);
97 4          If (((BLOCK0 AND OVEN$ON$MASK(n))<>0)
              AND (TEMP(n)<>0))
              then do;
99 5              STATUS(n)=7;
100 5              BLOCK2=BLOCK2 OR OVEN$RUN(n);
/* Test for an intolerance condition */
101 5              If SETPOINT(n)-TOLERANCE(n) < TEMP(n) AND
                  SETPOINT(n)+TOLERANCE(n) > TEMP(n)
                  then do;
103 6                  STATUS(n)=7;
104 6                  BLOCK1=BLOCK1 OR OVEN$INSTOL(n);
105 6              end;
106 5              else BLOCK1=BLOCK1 AND NOT OVEN$INSTOL(n);
/* Test for a caution condition */
107 5              If SETPOINT(n)-TOLERANCE(n) > T$t5(n) OR
                  SETPOINT(n)+TOLERANCE(n) < T$t5(n)
                  then do;
109 6                  STATUS(n)=14;
110 6                  BLOCK1=BLOCK1 OR OVEN$CAUTION(n);
111 6              end;
112 5              else BLOCK1=BLOCK1 AND NOT OVEN$CAUTION(n);
/* Test for a danger condition */
113 5              If SETPOINT(n)-TOLERANCE(n) > TEMP(n) OR
                  SETPOINT(n)+TOLERANCE(n) < TEMP(n)
                  then do;
115 6                  STATUS(n)=21;
116 6                  BLOCK2=BLOCK2 OR OVEN$DANGER(n);
117 6              end;
118 5              else BLOCK2=BLOCK2 AND NOT OVEN$DANGER(n);
/* Handle control of heater elements */
119 5              If SETPOINT(n) > T$t10(n)
                  then BLOCK3=BLOCK3 OR OVEN$HEATER(n);
121 5              else BLOCK3=BLOCK3 AND NOT OVEN$HEATER(n);
122 5              end;
123 4              else do;
/* Turn everything off when operator shuts off oven */
124 5                  BLOCK1=BLOCK1 AND NOT OVEN$INSTOL(n);
125 5                  BLOCK1=BLOCK1 AND NOT OVEN$CAUTION(n);
126 5                  BLOCK3=BLOCK3 AND NOT OVEN$HEATER(n);

```



```

127 5          BLOCK2=BLOCK2 AND NOT OVEN$DANGER(n);
128 5          BLOCK2=BLOCK2 AND NOT OVEN$RUN(n);
129 5          STATUS(n)=0;
130 5          end;
131 4          Call RQSEND(.CONSTANT$LOCKOUT$EXCH,MSG$PTR);
132 4          end;

```

```

/* Output data to real world */
133 3          OUTPUT(232)=BLOCK1;
134 3          OUTPUT(233)=BLOCK2;
135 3          OUTPUT(234)=BLOCK3;
136 3          end;
137 2          end CONTROL$TASK;
138 1          end CONTROL$TASK$MODULE;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0946H 2374D
VARIABLE AREA SIZE  = 0F54H 24D
MAXIMUM STACK SIZE  = 0006H 6D
235 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

```

$TITLE('CRT PARAMETER TASK')
/*****
* This task is used to examine and update the
* temperature setpoints and tolerances for
* each of the four ovens.
*****/

```

```

1          UPDATE$TASK:
          Do;

          $Include (:F0:COMMON.ELT)
2 1 =      DECLARE TRUE LITERALLY 'OFFH';
3 1 =      DECLARE FALSE LITERALLY 'OFFH';
4 1 =      DECLARE BOOLEAN LITERALLY 'BYTE';
5 1 =      DECLARE FOREVER LITERALLY 'WHILE 1';
          $Include (:F0:MSGTYP.ELT)
6 1 =      DECLARE DATA$TYPE LITERALLY '0',
          =      INT$TYPE LITERALLY '1',
          =      MISSED$INT$TYPE LITERALLY '2',
          =      TIME$OUT$TYPE LITERALLY '3',
          =      FSSREQ$TYPE LITERALLY '4',
          =      UC$REQ$TYPE LITERALLY '5',
          =      FSSNAK$TYPE LITERALLY '6',
          =      CNTRL$C$TYPE LITERALLY '7',
          =      READ$TYPE LITERALLY '8',
          =      CLR$RD$TYPE LITERALLY '9',
          =      LAST$RD$TYPE LITERALLY '10',
          =      ALARM$TYPE LITERALLY '11',
          =      WRITE$TYPE LITERALLY '12';
          $Include (:F0:MSG.ELT)

```

```

7 1 = DECLARE MSG$HDR LITERALLY
    = LINK ADDRESS,
    = LENGTH ADDRESS,
    = TYPE BYTE,
    = HOME$EX ADDRESS,
    = RESP$EX ADDRESS';

8 1 = DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE (
    = MSG$HDR,
    = REMAINDER(1) BYTE)';
    $Include (:F0:THMSG.ELT)
9 1 = DECLARE TH$MSG LITERALLY 'STRUCTURE (
    = MSG$HDR,
    = STATUS ADDRESS,
    = BUFFER$ADR ADDRESS,
    = COUNT ADDRESS,
    = ACTUAL ADDRESS,
    = REMAINDER(128) BYTE)';
10 1 = DECLARE MIN$TH$MSC$LENGTH LITERALLY '17';
    $Include (:F0:CHAR.ELT)

    = /* SPECIAL ASCII CHARACTERS */
    =
11 1 = DECLARE
    = NULL LITERALLY '0CH',
    = CONTROL$C LITERALLY '03H',
    = CONTROL$E LITERALLY '05H',
    = BELL LITERALLY '07H',
    = TAB LITERALLY '09H',
    = LF LITERALLY '0AH',
    = VT LITERALLY '0BH',
    = FF LITERALLY '0CH',
    = CR LITERALLY '0DH',
    = CONTROL$P LITERALLY '0EH',
    = CONTROL$Q LITERALLY '0FH',
    = CONTROL$R LITERALLY '10H',
    = CONTROL$S LITERALLY '11H',
    = CONTROL$T LITERALLY '12H',
    = CONTROL$U LITERALLY '13H',
    = CONTROL$X LITERALLY '18H',
    = CONTROL$Z LITERALLY '1AH',
    = ESC LITERALLY '1BH',
    = QUOTE LITERALLY '22H',
    = LCA LITERALLY '51H',
    = LCZ LITERALLY '7AH',
    = RUBOUT LITERALLY '7FH';
    =
    $Include (:F0:SYNCH.EXT)
12 1 = RQSEND:
    = PROCEDURE (EXCHANGE$POINTER, MESSAGE$POINTER) EXTERNAL;
13 2 = DECLARE (EXCHANGE$POINTER, MESSAGE$POINTER) ADDRESS;
    =
14 2 = END RQSEND;
    =
15 1 = RQWAIT:
    = PROCEDURE (EXCHANGE$POINTER, DELAY) ADDRESS EXTERNAL;
16 2 = DECLARE (EXCHANGE$POINTER, DELAY) ADDRESS;
    =

```

```

17 2 = END RQWAIT;
18 1 = RQACPT:
    = PROCEDURE (EXCHANGES$POINT,R) ADDRESS EXTERNAL;
19 2 = DECLARE EXCHANGES$POINTER ADDRESS;
20 2 = END RQACPT;
21 1 = RQISND:
    = PROCEDURE (IED$PTR) EXTERNAL;
22 2 = DECLARE IED$PTR ADDRESS;
23 2 = END RQISND;
24 1 DECLARE TEMP$CALIBRATE(5) address external;
25 1 DECLARE UPDATE$EXCH(5) address external;
26 1 DECLARE CRT$STATUS$EXCH(5) address external;
27 1 DECLARE COMP$EXCH(5) address external;
28 1 DECLARE CONSTANT$LOCKOUT$EXCH(5) address external;
29 1 DECLARE RQOUTX(5) address external;
30 1 DECLARE RQINPX(5) address external;
31 1 DECLARE WORDS$EXCH(5) address external;
32 1 DECLARE SETPOINT(4) address external;
33 1 DECLARE TOLERANCE(4) address external;
34 1 DECLARE BUFFER2 address;
35 1 DECLARE MSG$PTR address;
36 1 DECLARE MSG structure (
    MSG$HDR,
    STATUS address,
    BUFFER$PTR address,
    COUNT address,
    ACTUAL address );
37 1 DECLARE CAL$TEMP structure (
    MSG$HDR,
    CAL address );
38 1 DECLARE UPD$MSG address;
39 1 DECLARE ENERGIZE based UPD$MSG structure (
    MSG$HDR,
    STATUS address,
    BUFFER$PTR address,
    COUNT address,
    ACTUAL address );
40 1 DECLARE ENABLE$MSG structure (
    MSG$HDR );
41 1 DECLARE BUFFER(20) byte;
42 1 DECLARE OVEN byte;
43 1 DEC$REP:
    Procedure (SOURCE,TARGET) external;
    Declare (SOURCE,TARGET) address;
44 2 end DEC$REP;
45 2 ASC$2$BINARY:
    Procedure (SOURCE,TARGET,SIZE) byte external;
47 2 Declare (SOURCE,TARGET) address;
48 2 Declare SIZE byte;
49 2 end ASC$2$BINARY;
50 1 Declare MSG$1(20) byte data (
    ESC,'E','ENTER OVEN NUMBER-');

```

```

      'XXXX.X-' );
52  1  Declare MSG$3(20) byte data (
      CR,LF,
      'ENTER NEW TOLERANCE-',
      'XXXX.X-' );
53  1  Declare CALMSC(12) byte data (
      'TEMPERATURE-' );
54  1  Declare MSG$4(62) byte data (
      CR,LF,
      '(STATUS-(S), PARAMETERS-(P), CALIBRATE-(C))',
      CR,LF,
      'ENTER REQUEST-' );
55  1  Declare WAIT literally 'MSG$PTR=';
56  1  Declare FOR literally 'RQWAIT';
57  1  Declare START literally 'CALL';
58  1  Declare TASK literally 'RQSEND';
59  1  UPDATE:
      Procedure public;
      /* Initialize task at start-up time */
60  2  Do forever;
61  3  MSG.RESP$EX=.COMP$EXCH;
      /* Wait for request to enter task */
62  3  UPD$MSG=RQWAIT (.UPDATE$EXCH,C);
      /* Get desired oven number from operator */
63  3  RQST$OVEN:
      MSG.BUFFER$PTR=.MSC$1;
64  3  MSG.TYPE=WRITE$TYPE;
65  3  MSG.COUNT=20;
66  3  Start task (.RQOUTX,.MSG);
67  3  Wait for (.COMP$EXCH,C);
      /* ...Input new number */
68  3  MSG.BUFFER$PTR=.BUFFER;
69  3  MSG.COUNT=255;
70  3  MSG.TYPE=CLR$RD$TYPE;
71  3  Start task (.RQINPX,.MSG);
72  3  Wait for (.COMP$EXCH,C);
73  3  OVEN=(BUFFER(0) AND 07H)-1;
74  3  If OVEN > 3 then go to RQST$OVEN;
      /* Display request and current setpoint */
76  3  GET$TEMP:
      Call move (28,.MSG$2,.BUFFER);
77  3  Call DEC$REP (.SETPOINT(oven),.BUFFER+21);
78  3  MSG.TYPE=WRITE$TYPE;
79  3  MSG.COUNT=28;
80  3  Start task (.RQOUTX,.MSG);
81  3  Wait for (.COMP$EXCH,C);
      /* ... Input new setpoint */
82  3  MSG.TYPE=CLR$RD$TYPE;
83  3  Start task (.RQINPX,.MSG);
84  3  Wait for (.COMP$EXCH,C);
85  3  If ASC$2$BINARY(.BUFFER,.BUFFER2,1)=0 OR BUFFER2 > 700
      then go to GET$TEMP;
87  3  If BUFFER2 <> 0
      then do;
89  4  Wait for (.CONSTANT$LOCKOUT$EXCH,C);
90  4  SETPOINT(oven)=BUFFER2;
91  4  Start task (.CONSTANT$LOCKOUT$EXCH,MSG$PTR);

```

```

93 3 GETSTOL:
94 3 Call move (29,.MSG$3,.BUFFER);
95 3 Call DEC$REP (.TOLERANCE(oven),.BUFFER+22);
96 3 MSG.TYPE=WRITE$TYPE;
97 3 MSG.COUNT=29;
98 3 Start task (.RQOUTX,.MSG);
99 3 /* ...Input new tolerance */
100 3 MSG.TYPE=CLR$RD$TYPE;
101 3 Start task (.RQINPX,.MSG);
102 3 Wait for (.COMP$EXCH,0);
103 3 If ASC$2$BINARY(.BUFFER,.BUFFER2,1)=0 OR BUFFER2 > 700
104 3 then go$to GETSTOL;
105 3 If BUFFER2 <> 0
106 3 then do;
107 4 Wait for (.CONSTANT$LOCKOUT$EXCH,0);
108 4 TOLERANCE(oven)=BUFFER2;
109 4 Start task (.CONSTANT$LOCKOUT$EXCH,MSG$PTR);
110 4 end;
111 3 /* Ask operator if he is finished */
112 3 REQ$NEXT:
113 3 MSG.TYPE=WRITE$TYPE;
114 3 MSG.COUNT=62;
115 3 MSG.BUFFER$PTR=.MSG$4;
116 3 Start task (.RQOUTX,.MSG);
117 3 Wait for (.COMP$EXCH,0);
118 3 /* ...Get his response */
119 3 MSG.TYPE=CLR$RD$TYPE;
120 3 MSG.BUFFER$PTR=.BUFFER;
121 3 Start task (.RQINPX,.MSG);
122 3 Wait for (.COMP$EXCH,0);
123 3 If (BUFFER(0) <> 'S' AND BUFFER(0) <> 'P'
124 3 AND BUFFER(0) <> 'C')
125 3 then go$to REQ$NEXT;
126 3 If BUFFER(0)='P'
127 3 then go$to RQST$OVEN;
128 3 If BUFFER(0)='C'
129 3 then do;
130 4 GET$CAL:
131 4 MSG.TYPE=WRITE$TYPE;
132 4 MSG.COUNT=12;
133 4 MSG.BUFFER$PTR=.CALMSG;
134 4 Start task (.RQOUTX,.MSG);
135 4 Wait for (.COMP$EXCH,0);
136 4 MSG.TYPE=CLR$RD$TYPE;
137 4 MSG.BUFFER$PTR=.BUFFER;
138 4 Start task (.RQINPX,.MSG);
139 4 Wait for (.COMP$EXCH,0);
140 4 If ASC$2$BINARY(.BUFFER,.BUFFER2,1) = 0
141 4 OR BUFFER2 > 250 OR BUFFER2 < 200
142 4 then go$to GET$CAL;
143 4 CAL$TEMP.CAL=BUFFER2;
144 4 Call RQSEND (.TEMP$CALIBRATE,.CAL$TEMP);
145 4 end;

```



```

MODULE INFORMATION:
CODE AREA SIZE = 03C3H 963D
VARIABLE AREA SIZE = 007CH 124D
MAXIMUM STACK SIZE = 0004H 4D
264 LINES READ
0 PROGRAM ERROR(S)
END OF PL/M-84 COMPILATION

139 3 ENERGIZE.TYPE=100;
140 3 Start task (.CRT$STATUS$EXCH,UPD$MSG);

141 3 end;
142 2 end UPDATE;
143 1 end UPDATE$TASK;

$TITLE('CRT UPDATE TASK')
/*****
* This task is utilized to update the CRT ter- *
* minal display with the current operating par- *
* ameters. It will be entered upon sytem start- *
* up, upon operator request, or when a problem *
* exists with any of the activated ovens. *
*****/
1 CRT$DATA$MODULE:
Do;
$INCLUDE(:F0:SYNCH.EXT)
2 1 = RQSEND:
3 2 = PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
4 2 = DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
5 2 = END RQSEND;
6 1 = RQWAIT:
7 2 = PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
8 2 = DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
9 2 = END RQWAIT;
10 1 = RQACPT:
11 2 = PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
12 2 = DECLARE EXCHANGE$POINTER ADDRESS;
13 2 = END RQACPT;
14 1 = RQISND:
15 2 = PROCEDURE (IED$PTR) EXTERNAL;
16 2 = DECLARE IED$PTR ADDRESS;
17 2 = END RQISND;
18 2 = $INCLUDE (:F0:MSGTYP.ELT)
19 1 = DECLARE DATA$TYPE LITERALLY '0';

```

```

=          INT$TYPE LITERALLY '1',
=          MISSED$INT$TYPE LITERALLY '2',
=          TIME$OUT$TYPE LITERALLY '3',
=          FSS$REQ$TYPE LITERALLY '4',
=          UCS$REQ$TYPE LITERALLY '5',
=          FSS$NAK$TYPE LITERALLY '6',
=          CNTRL$C$TYPE LITERALLY '7',
=          READ$TYPE LITERALLY '8',
=          CLR$RD$TYPE LITERALLY '9',
=          LAST$RD$TYPE LITERALLY '10',
=          ALARM$TYPE LITERALLY '11',
=          WRITE$TYPE LITERALLY '12',
=          $INCLUDE (:F0:EXCH.ELT)
15 1 = DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
=          MESSAGE$HEAD ADDRESS,
=          MESSAGE$TAIL ADDRESS,
=          TASK$HEAD ADDRESS,
=          TASK$TAIL ADDRESS,
=          EXCHANGE$LINK ADDRESS)';
=          $INCLUDE (:F0:COMMON.ELT)
16 1 = DECLARE TRUE LITERALLY 'OFFH';
17 1 = DECLARE FALSE LITERALLY 'OOH';
18 1 = DECLARE BOOLEAN LITERALLY 'BYTE';
19 1 = DECLARE FOREVER LITERALLY 'WHILE';
=          $INCLUDE (:F0:MSG.ELT)
20 1 = DECLARE MSG$HDR LITERALLY '
=          LINK ADDRESS,
=          LENGTH ADDRESS,
=          TYPE BYTE,
=          HOME$EX ADDRESS,
=          RESP$EX ADDRESS';
=          DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE (
21 1 =          MSG$HDR,
=          REMAINDER(1) BYTE)';
=          $INCLUDE (:F0:CHAR.ELT)
=          /* SPECIAL ASCII CHARACTERS */
=          DECLARE
22 1 =          NULL LITERALLY 'OOH';
=          CONTROL$C LITERALLY '03H';
=          CONTROL$E LITERALLY '05H';
=          BELL LITERALLY '07H';
=          TAB LITERALLY '09H';
=          LF LITERALLY '0AH';
=          VT LITERALLY '0BH';
=          FF LITERALLY '0CH';
=          CR LITERALLY '0DH';
=          CONTROL$P LITERALLY '10H';
=          CONTROL$Q LITERALLY '11H';
=          CONTROL$R LITERALLY '12H';
=          CONTROL$S LITERALLY '13H';
=          CONTROL$X LITERALLY '18H';
=          CONTROL$Z LITERALLY '1AH';
=          ESC LITERALLY '1BH';
=          QUOTE LITERALLY '22H';

```



```

44 1 Declare OVENS0N(4) byte data (
    01H,02H,04H,08H );
45 1 Declare OVENS0CAUTION(4) byte data (
    10H,20H,40H,80H );
46 1 Declare CRT structure (
    MSG$HDR,
    STATUS address,
    BUFFER$PTR address,
    COUNT address,
    ACTUAL address );
47 1 Declare CRTLOCK structure (MSG$HDR);
48 1 Declare CRT$DISPLAY$LOCK(5) address external;
49 1 Declare TEMP$LOCKOUT$EXCH(5) address external;
50 1 Declare CONSTANT$LOCKOUT$EXCH(5) address external;
51 1 Declare CRT$EXCH(5) address external;
52 1 Declare CRT$STATUS$EXCH(5) address external;
53 1 Declare DUMMY$EXCH(5) address external;
54 1 Declare READ$BUFFER$EXCH(5) address external;
55 1 Declare UPDATE$EXCH(5) address external;
56 1 Declare RQINPX(5) address external;
57 1 Declare RQOUTX(5) address external;
58 1 Declare RQWAKE(5) address external;
59 1 Declare RQL7EX(5) address external;
60 1 Declare RQL6EX(5) address external;
61 1 Declare RQDBUG(5) address external;
62 1 Declare RQALRM(5) address external;
63 1 Declare TEMP(4) address external;
64 1 Declare DISP$TEMP(4) address;
65 1 Declare SETPOINT(4) address external;
66 1 Declare DISP$SETPNT(4) address;
67 1 Declare TOLERANCE(4) address external;
68 1 Declare DISP$TOL(4) address;
69 1 Declare STATUS(4) byte external;
70 1 Declare DISP$STAT(4) byte;
71 1 Declare (BLOCK1,BLOCK2) byte external;
72 1 Declare WORK$BUFF(170) byte;
73 1 Declare BUFFER$A(70) byte;
74 1 Declare (CHANGE,n,ALARM,NEW,BLANKER) byte;
75 1 Declare START literally 'call';
76 1 Declare TASK literally 'rqsend';
77 1 Declare WAIT literally 'msg$ptr=';
78 1 Declare For literally 'rqwait';
79 1 DEC$REP:
    Procedure(SOURCE,TARGET) external;
80 2 Declare (SOURCE,TARGET) address;
81 2 end DEC$REP;

```



```

82 1 CRT$DATA$TASK:PROC (.RQOUTX, .CRT$EXCH, .CRT$LOCKOUT$EXCH, .STARTER(0));
Procedure public;
/* Initialize system at start-up time */
83 2 Start task (.TEMP$LOCKOUT$EXCH, .STARTER(0));
84 2 Start task (.CONSTANT$LOCKOUT$EXCH, .STARTER(1));
85 2 STARTER(2).TYPE=100;
86 2 Start task (.CRT$STATUS$EXCH, .STARTER(2));
87 2 CRT.RESP$EX=.CRT$EXCH;
/* Perform main CRT wait */
88 2 Do forever;
89 3 Wait for (.DUMMY$EXCH, 10);
90 3 Wait for (.CRT$STATUS$EXCH, 0);
91 3 If MSG.TYPE=255
then ALARM=1;
else ALARM=0;
/* Output heading */
94 3 If (MSG.TYPE=100 OR MSG.TYPE=255)
then do;
96 4 If ALARM=0
then call ROSEND (.CRT$DISPLAY$LOCK, .CRT$LOCK);
98 4 CRT.TYPE=WRITE$TYPE;
99 4 CRT.COUNT=167;
100 4 CRT.BUFFER$PTR=.WORK$BUFF;
101 4 READ.TYPE=CLR$RD$TYPE;
102 4 READ.COUNT=255;
103 4 READ.RESP$EX=.READ$BUFFER$EXCH;
104 4 READ.BUFFER$PTR=.BUFFERA;
105 4 If ALARM=0
then start task (.RQINPX, .READ);
Call move (82, .CRT$HDR, .WORK$BUFF);
Call move (86, .CRT$HDR+82, .WORK$BUFF+82);
Start task (.RQOUTX, .CRT);
Wait for (.CRT$EXCH, 0);
NEW=1;
end;
/* Test for change in temperature of any oven */
113 3 CHANGE=0;
114 3 Wait for (.TEMP$LOCKOUT$EXCH, 0);
115 3 Do n=0 to 3;
116 4 If TEMP(n) <> DISP$TEMP(n)
then CHANGE=1;
118 4 end;
119 3 Call move (8, .TEMP, .DISP$TEMP);
120 3 Start task (.TEMP$LOCKOUT$EXCH, MSG$PTR);
/* When a change exists build new line */
121 3 If CHANGE OR NEW
then do;
123 4 Call move (90, .L1$IMAGE, .WORK$BUFF);
124 4 Do n=0 to 3;
125 5 Call DEC$REP (.DISP$TEMP(n), DISPLAY$PTR1(n));
126 5 end;
/* Output new temperature line to CRT */
127 4 CRT.TYPE=WRITE$TYPE;
128 4 CRT.COUNT=87;

```

```

129 4          Start task (.RQOUTX,.CRT);
130 4          Wait for (.CRT$EXCH,0);
131 4          end;
/* Test for change in oven setpoints */
132 3  CHANGE=0;
133 3  Wait for (.CONSTANT$LOCKOUT$EXCH,0);
134 3  Do n=0 to 3;
135 4      If SETPOINT(n) <> DISP$SETPNT(n)
        then CHANGE=1;
137 4      end;
138 3      Call move (8,.SETPOINT,.DISP$SETPNT);
139 3      Start task (.CONSTANT$LOCKOUT$EXCH,MSG$PTR);
/* Build new line when a change was detected */
140 3      If CHANGE OR NEW
        then do;
142 4          Call move (92,.L2$IMAGE,.WORK$BUFF);
143 4          Do n=0 to 3;
144 5              Call DEC$REP(.DISP$SETPNT(n),DISPLAY$PTR2(n));
145 5          end;
/* Output setpoint line */
146 4      CRT.TYPE=WRITE$TYPE;
147 4      CRT.COUNT=89;
148 4      CRT.BUFFER$PTR=.WORK$BUFF;
149 4      Start task (.RQOUTX,.CRT);
150 4      Wait for (.CRT$EXCH,0);
151 4      end;
/* Test for change in tolerance line */
152 3  CHANGE=0;
153 3  Wait for (.CONSTANT$LOCKOUT$EXCH,0);
154 3  Do n=0 to 3;
155 4      If TOLERANCE(n) <> DISP$TOL(n)
        then CHANGE=1;
157 4      end;
158 3      Call move (8,.TOLERANCE,.DISP$TOL);
159 3      Start task (.CONSTANT$LOCKOUT$EXCH,MSG$PTR);
/* When change is found, build new line */
160 3      If CHANGE OR NEW
        then do;
162 4          Call move (94,.L3$IMAGE,.WORK$BUFF);
163 4          Do n=0 to 3;
164 5              Call DEC$REP(.DISP$TOL(n),DISPLAY$PTR2(n));
165 5          end;
/* Output tolerance line */
166 4      CRT.TYPE=WRITE$TYPE;
167 4      CRT.COUNT=91;
168 4      CRT.BUFFER$PTR=.WORK$BUFF;
169 4      Start task (.RQOUTX,.CRT);
170 4      Wait for (.CRT$EXCH,0);
171 4      end;
/* Build status message */
172 3      CHANGE=0;
173 3      Wait for (.CONSTANT$LOCKOUT$EXCH,0);
174 3      Do n=0 to 3;
175 4          If STATUS(n) <> DISP$STAT(n)
        then CHANGE=1;

```

```

177 4      end;
178 3      Call move (4,.STATUS,.DISP$STAT);
179 3      Start task (.CONSTANT$LOCKOUT$EXCH,MSG$PTR);
      /* Output to display */
180 3      If CHANGE OR NEW
      then do;
182 4          Call move (75,.L4IMAGE,.WORK$BUFF);
183 4          Do n=F to 3;
184 5              Call move(7,.MESSAGES+DISP$STAT(n),DISPLAY$PTR4(
n));
185 5          end;
186 4          CRT.COUNT=75;
187 4          Start task (.RQOUTX,.CRT);
188 4          Wait for (.CRT$EXCH,0);
189 4      end;
      /* test for request to exit this mode */
190 3      MSG$PTR=RQACPT (.READ$BUFFER$EXCH);
191 3      If ALARM=0
      then do;
193 4          If (MSG$PTR <> 0 and BUFFERA(0) = 1FH)
      then do;
195 5              MSC$PTR=RQWAIT (.CRT$DISPLAY$LOCK,0);
196 5              start task (.UPDATE$EXCH,MSG$PTR);
197 5          end;
198 4          else do;
199 5              If MSG$PTR=0
      then STARTER(2).TYPE=200;
      else STARTER(2).TYPE=100;
      Start task (.CRT$STATUS$EXCH,.STARTER(2));
201 5              NEW=0;
202 5          end;
203 5          end;
204 5      end;
205 4      end;
206 3      end;
207 2      end CRT$DATA$TASK;
208 1      end CRT$DATA$MODULE;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0720H  1824D
VARIABLE AREA SIZE  = 0189H   393D
MAXIMUM STACK SIZE  = 0004H    4D
388 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

```

$TITLE('ASCII STRING TO FIXED BINARY')
/*****
* This program converts an ASCII string into a fixed point binary number. The fixed decimal point
* is determined by the parameter passed in SIZE.
*****/
1  ASC$2$BINARY$MODULE:
  Do;
  /* SPECIAL ASCII CHARACTERS */
  2  1  DECLARE
        NULL          LITERALLY '00H',
        CONTROL$C      LITERALLY '03H',
        CONTROL$E      LITERALLY '05H',
        BELL           LITERALLY '07H',
        TAB            LITERALLY '09H',
        LF             LITERALLY '0AH',
        VT             LITERALLY '0BH',
        FF             LITERALLY '0CH',
        CR             LITERALLY '0DH',
        CONTROL$P      LITERALLY '10H',
        CONTROL$Q      LITERALLY '11H',
        CONTROL$R      LITERALLY '12H',
        CONTRCL$S      LITERALLY '13H',
        CONTROL$X      LITERALLY '18H',
        CONTROL$Z      LITERALLY '1AH',
        ESC            LITERALLY '1BH',
        QUOTE          LITERALLY '22H',
        LCA            LITERALLY '51H',
        LCZ            LITERALLY '7AH',
        RUBOUT         LITERALLY '7FH';

  3  1  ASC$2$BINARY:
        Procedure (SRC$PTR,TRGT$PTR,SIZE) byte public;
        4  2  Declare (SRC$PTR,TRGT$PTR) address;
        5  2  Declare (SOURCE based SRC$PTR) (30) byte;
        6  2  Declare RESULT based TRGT$PTR address;
        7  2  Declare (N,SIZE,K,DP,DIGITS,VALID) byte;
        8  2  Declare POWER(6) address data (
              0,1,10,100,1000,10000 );
        /* Find location of decimal point */
        9  2  n=0;
        10 2  Do while SOURCE(n)<>'.' AND SOURCE(n)<>CR
              AND SOURCE(n)<>LF;
        11 3  n=n+1;
        12 3  end;
        13 2  DP=n;
        /* Provide correct number of digits to right of decimal */
        14 2  Do n=0 to SIZE;
        15 3  SOURCE(DP+n)=SOURCE(DP+n+1);
        16 3  If SOURCE(DP+n)>39H OR SOURCE(DP+n)<30H
              then do k=n to SIZE;
        18 4  SOURCE(DP+k)='0';
        19 4  end;

```

```

20 3      end;
21 2      DIGITS=DP+SIZE;
22 2      /* Test for all valid characters */
23 2      VALID=1;
24 3      Do n=0 to DIGITS;
25 4          If SOURCE(n)>39H OR SOURCE(n)<30H
26 5              then VALID=0;
27 2      end;
28 2      If DIGITS>5
29 3          then VALID=0;
30 2      /* Convert data to binary and store */
31 2      n=0;
32 2      If VALID=1
33 3          then do;
34 4              RESULT=0;
35 4              Do while DIGITS > 0;
36 5                  RESULT=RESULT+((
37 6                      SOURCE(n) AND 0FH) * POWER(DIGITS));
38 5                  n=n+1;
39 5                  DIGITS=DIGITS-1;
40 4              end;
41 2          end;
42 2      /* Return to calling program */
43 2      Return VALID;
44 2      end ASC$2$BINARY;
45 1      end ASC$2$BINARY$MODULE;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0178H      376D
VARIABLE AREA SIZE = 000AH      10D
MAXIMUM STACK SIZE = 0004H      4D
80 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

```

$TITLE('COMMON VARIABLE STORAGE')
/*****
* This module contains those variables common to
* multiple tasks in the oven control application.
*****/
1 VARIABLE$STORAGE:
2   Do;
3   1 Declare SETPOINT(4) address public;
4   1 Declare TOLERANCE(4) address public;
5   1 Declare TEMP(4) address public;
6   1 Declare STATUS(4) byte public;
7   1 Declare BLOCK0 byte public;
8   1 Declare BLOCK1 byte public;
9   1 Declare BLOCK2 byte public;
10  1 Declare BLOCK3 byte public;
    end VARIABLE$STORAGE;

MODULE INFORMATION:
    CODE AREA SIZE      = 0000H
    VARIABLE AREA SIZE  = 0020H
    MAXIMUM STACK SIZE  = 0000H
    16 LINES READ
    0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION

```

```

MODULE INFORMATION:
CODE AREA SIZE      = 0000H
VARIABLE AREA SIZE  = 0020H
MAXIMUM STACK SIZE  = 0000H
16 LINES READ
0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION

```



```

$TITLE('WORD TO ASCII CONVERSION')
/*****
* This routine converts a fixed point word in mem- *
* ory into a 4 digit plus 1 decimal ASCII display- *
* able number. Zero blanking is included. *
*****/
1  DEC$REP$MODULE:
   Do;

2  1  DEC$REP:
   Procedure (SOURCE,TARGET) public ;
3  2      Declare (SOURCE,TARGET) address;
4  2      Declare ANSWR(5) byte;
5  2      Declare (DISPLAY based TARGET)(5) byte;
6  2      Declare NUMBER based SOURCE structure (
           ELEMENT address );
7  2      Declare N byte;
8  2      Declare CALC(5) address;
   /* Initialize */
9  2      Do n=0 to 4;
10 3          ANSWR(n)='0';
11 3      end;
12 2      CALC(0)=NUMBER.ELEMENT;
   /* Convert to ASCII */
13 2      Do n=1 to 5;
14 3          CALC(n)=CALC(n-1)/10;
15 3          ANSWR(5-n)=(CALC(n-1) mod 10) + 30H;
16 3      end;
   /* Perform zero blanking */
17 2      Do n=0 to 3;
18 3          If ANSWR(n)<>'0'
20 3              then n=4;
21 3              else ANSWR(n)=' ';
22 3      end;
   /* Format with decimal point */
23 2      Call move (4,.ANSWR,TARGET);
24 2      DISPLAY(4)='.';
25 2      DISPLAY(5)=ANSWR(4);
26 1  end DEC$REP;
   end DEC$REP$MODULE;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 00EEH      238D
VARIABLE AREA SIZE = 0014H      20D
MAXIMUM STACK SIZE = 0004H      4D
40 LINES READ
0 PROGRAM ERROR(S)
END OF PL/M-80 COMPILATION

```


November 1979

CONTROLLER	3-66
Input/Output Functions	3-66
IV. APPLICATION EXAMPLE	3-67
Mechanical Specifications	3-69
Interface Requirements	3-70
Weightbelt Weight	3-70
Weightbelt Motor Control	3-71
Weightbelt Speed Measurement	3-72
Liquid Flow Control	3-73
Liquid Flow Measurement	3-73
Operator Interface	3-74
Interface Summary	3-74
V. HARDWARE CONFIGURATION	3-74
Controller Interface	3-75
VI. SOFTWARE CONFIGURATION	3-75
High Level Programming Languages	3-80
Fundamental Support Packages	3-80
Host/Slave Relationship	3-80
RMX/80 BASIC-80 Interpreter	3-81
Software Tasks	3-81
VII. SOFTWARE DRIVERS	3-81
Motor Speed Control Processor	3-81
Weightbelt Control Processor	3-82
Stepper Control Processor	3-87
VIII. APPLICATION SOFTWARE	3-90
Application Software	3-90
Control Software	3-91
IX. CONCLUSION	3-92
APPENDIX A	3-92

Closed Loop Control Using The iSBC 569/941™ Intelligent Digital Processors

Peter Andersen
OEM Microcomputer Systems Applications

**Closed Loop Control
Using the iSBC 569/941
Intelligent Digital
Processors**

Contents

I. INTRODUCTION	3-63
Reasons for Intelligent Boards	3-63
The On-Board Slave Concept	3-63
II. BASIC UNIVERSAL PERIPHERAL INTERFACE DISCUSSIONS	3-64
Hardware Features	3-64
Software Interface	3-64
Standard Universal Peripheral Controllers	3-65
Industrial Digital Processor	3-66
III. FUNCTIONS OF THE INTELLIGENT DIGITAL CONTROLLER	3-66
Input/Output Functions	3-66
IV. APPLICATION EXAMPLE	3-67
Mechanical Specifications	3-69
Interface Requirements	3-70
Weightbelt Weight	3-70
Weightbelt Motor Control	3-71
Weightbelt Speed Measurement	3-72
Liquid Flow Control	3-72
Liquid Flow Measurement	3-73
Operator Interface	3-74
Interface Summary	3-74
V. HARDWARE CONFIGURATION	3-74
Controller Interface	3-75
VI. SOFTWARE CONFIGURATION	3-79
High Level Programming Languages ..	3-80
Fundamental Support Packages	3-80
Host/Slave Relationship	3-80
RMX/80 BASIC-80 Interpreter	3-81
Software Tasks	3-81
VII. SOFTWARE DRIVERS	3-81
Motor Speed Control Processor	3-81
Weight Input Processor	3-85
Stepper Motor Control Processor	3-87
VIII. APPLICATION SOFTWARE	3-90
Initialization Programs	3-90
Control Algorithm Programs	3-91
Master Processor	3-91
IX. CONCLUSION	3-92
APPENDIX A	3-95

I. INTRODUCTION

The utilization of computers to provide control or monitoring functions for industrial processes frequently results in complex computer systems. Distributing the control and processing intelligence throughout the control network reduces significantly the complexity of the system while increasing the reliability. The physical areas being controlled or monitored by each portion of the distributed system will generally consist of a relatively small number of I/O functions which are related by some control algorithm.

The Intel iSBC 569 Intelligent Digital Controller (IDC) and the iSBC 941 Industrial Digital Processor (IDP) are a part of the expanding line of Intel products which are oriented toward filling the requirements of these systems. This application note deals with the use of these devices to provide control of a closed loop system using a version of the PID control algorithm.

It is assumed that the reader is familiar with the basic concepts required to generate software and has had some experience with using a computer. This application note will then guide the reader through a typical application, explaining in detail the decisions which must be made in order to effectively utilize a microcomputer to provide a control solution.

The application which has been chosen is considered to be typical of the type which lends itself to control. The mechanical aspects of the application will be explained so that the user not familiar with the particular machinery will be able to understand the development. It will be seen that the techniques used will apply to any other specific application.

The emphasis of the note will be on the use and implementation of the hardware and software features of the digital processor and controller. The actual PID control algorithm will not be developed in this application note.

Reasons for Intelligent Boards

The advent of microcomputers and the resulting trend toward utilizing these devices to control processes has resulted in many cases where the overall system performance has deteriorated because of the demands placed on the processor.

In these applications, the computer has become overburdened with control algorithms, alarm detection, communications, and the many other tasks required of it. The processor can be interrupted by time dependent tasks to the point where other processing tasks can not be completed.

Presently, Intel provides two I/O expansion boards which are capable of handling portions of the processing load which formally required processor time. These two devices are the iSBC 544 Intelligent Communications Controller and the iSBC 569 Intelligent Digital Controller. Tasks which involve communications or parallel digital I/O can now be offloaded without requiring valuable processor time. These boards can issue interrupts to the master or host processor if interaction with other processes or devices is required. This technique greatly increases system throughput by offloading the other bus master processors and by minimizing traffic on the Multibus system bus.

In some cases, it will be found that the intelligent controller can function to control the process in a stand-alone environment, providing a more functional, low cost control system.

The concept of offloading the processor of its input/output tasks can be developed on the iSBC 569 controller through the use of slave processors which may be installed on the board to assist the host. The result is the ability to provide up to four processors on a single intelligent slave I/O board by using the concept of slave processors.

The On-Board Slave Concept

The utilization of the iSBC 569 controller is enhanced through the use of On Board Slave processors (OBS). These devices distribute the system intelligence and offload the processor on the intelligent controller. They can provide custom digital interfaces with the various devices which may be connected to the I/O ports of the controller. The OBS device allows a designer to fully specify his control/interface algorithm in the peripheral chip without relying on the master processor. Three types of OBS compatible devices are available from Intel. These are: 1) Industrial Processors, 2) Standard UPI devices, and 3) UPI 8741A for custom applications. By combining the

devices in various combinations, optimum solutions can be generated for different control applications.

Before proceeding, we should cover the general characteristics of the OBS devices available for use in conjunction with the iSBC 569 controller. It will be seen that careful selection of the proper I/O controller chip can reduce significantly the design effort required to provide a control solution.

II. BASIC UNIVERSAL PERIPHERAL INTERFACE DISCUSSION

With the introduction of the Universal Peripheral Interface, Intel has expanded the intelligent peripheral concept by providing an intelligent controller that is fully user programmable. The 8741A is a complete single-chip microcomputer which connects directly to a master processor data bus.

To fully understand the techniques used by the UPI 8741A devices, we must have a general knowledge of their characteristics. Only then will we feel comfortable in implementing a design which uses the components.

Hardware Features

Each Universal Peripheral Interface has 1K bytes of program storage plus 64 bytes of RAM memory for data storage. It has a powerful, 8-bit CPU with a 2.5 μ sec cycle time and two interrupts. Over 90 instructions are provided in its instruction set. Most instructions are single byte and single cycle and none are more than two bytes long. These instructions are optimized for bit manipulation and I/O operations. Special instructions are included to allow binary or BCD arithmetic operations, table lookup routines, loop counters, and N-way branch routines.

The chip's 8-bit interval timer/event counter can be used to generate complex timing sequences for control applications or it can count external events such as switch closures and position encoder pulses. Software timing loops can be simplified or eliminated by the interval timer. If enabled, an interrupt to the CPU can occur when the timer overflows.

Two 8-bit bidirectional I/O ports are included which are TTL compatible. Each of the 16 port

lines can individually function as either input or output under software control.

The UPI microcomputer is fully supported with development tools. The combination of device features and Intel development support make the 8741A an ideal component for low-speed peripheral control applications.

Software Interface

The OBS communicates with the processor on the host board by means of data transfers between its registers and the host board's data bus. A communication protocol has been defined which provides a set of rules by which the processors may interact with each other. Two types of software protocol are currently defined. These are the "simple" and the "extended" protocol. Before attempting to utilize the OBS devices in an application, the concepts used for the communications must be fully understood.

When used on one of Intel's single board computers, the communication path is by means of the I/O ports on the host board. This means that two port addresses, an odd and an even location, are assigned to each OBS device. The even numbered port is used to transfer "data" between the processors. The odd numbered port is used to write commands into the OBS and to read its status. Each transfer between the host and the slave device consists of the movement of eight bits of information.

Four of the eight bits available in the status message have been given predefined functions. The bit will be set (logical 1) when the corresponding condition exists within the OBS device and will be reset (logical 0) when the condition does not exist. The functions of the four bits are:

Bit-0. Output Buffer Full (OBF).

This bit indicates that the OBS has placed information into the transfer register and that the information is available to the host processor. It can be read by performing an input operation from the even numbered port assigned to the particular OBS. When the data has been read, the bit will automatically be reset to indicate that no data is available. As we will see, this is one of the key features enabling efficient utilization of the host/

slave relationships on single board computers.

Bit-1. Input Buffer Full (IBF).

This bit is used to indicate that data has been placed into the input transfer register by the host device and that it has not yet been read by the slave. Data is transferred into the input register by means of the host performing an output to the even numbered port of the OBS. The bit will be reset when the device reads the data from the input transfer register into its accumulator. Data should only be output to the OBS when the IBF bit is reset!

Bit-2. F0 Flag.

Unlike the IBF and the OBF bits which are controlled by hardware, the F0 bit is controlled by the device software. The normal function of the flag is to provide a lockout to prevent the host from sending more data until the previous data has been processed or the operation is complete.

Bit-3. F1 is the Command/Data Flag.

It is automatically set when the host sends either a command (odd numbered port) or data (even numbered port). A logical 1 indicates that a command has been sent and a logical 0 indicates that data has been sent. This bit may also be cleared or toggled by the UPI software.

These bits will provide normal communications between the master and slave processors.

Figure 1 shows the sequence of operations which can be used by the host processor to establish communications with an OBS using the simple protocol. In Figure 1a, we see that all operations are initiated by the host. It will first verify that the IBF flag indicates that the input register is empty and available for receiving a command. The command is then sent to the odd numbered port. This command will inform the OBS that it is to perform some task. The task may involve a requirement for more information to be sent to the controller and it may involve the controller returning some data to the host. Figure 1b shows the operations required for receiving data from the OBS.

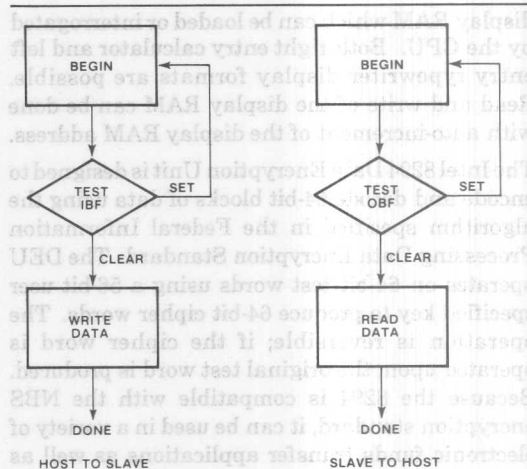


Figure 1. Simple Protocol

With these ideas in mind, we can move to a discussion of representative versions of the devices available for use on the IDC boards. We will then look at a typical application to see how they can actually be applied to solve a problem.

Standard Universal Peripheral Controllers

Intel presently manufactures three UPI controllers for non-industrial applications. These are:

1. 8278 Programmable Keyboard Interface
2. 8294 Data Encryption Unit
3. 8295 Dot Matrix Printer Controller

These devices offer an "off the shelf" solution to many applications which might be encountered.

The Intel 8278 is a general purpose programmable keyboard and display interface device. The keyboard portion can provide a scanned interface to 128-key contact or capacitive-coupled keyboards. The keys are fully debounced with N-key rollover and programmable error generation on multiple new key closures. Keyboard entries are stored in an 8-character FIFO with overrun status indication when more than 8-characters have been entered. Key entries set an interrupt request output to the master CPU. The display portion of the 8278 provides a scanned display interface for LED, incandescent, and other popular display technologies. Both numeric displays and simple indicators may be used. The 8278 has a 16 x 4

display RAM which can be loaded or interrogated by the CPU. Both right entry calculator and left entry typewriter display formats are possible. Read and write of the display RAM can be done with auto-increment of the display RAM address.

The Intel 8294 Data Encryption Unit is designed to encode and decode 64-bit blocks of data using the algorithm specified in the Federal Information Processing Data Encryption Standard. The DEU operates on 64-bit test words using a 56-bit user specified key to produce 64-bit cipher words. The operation is reversible; if the cipher word is operated upon, the original test word is produced. Because the 8294 is compatible with the NBS encryption standard, it can be used in a variety of electronic funds transfer applications as well as other electronic banking and data handling applications where data must be encrypted.

Finally, the Intel 8295 Dot Matrix Printer Controller provides an interface to the LRC 7040 Series dot matrix impact printers. It may also be used as an interface to other similar printers. The chip may be used in a serial or parallel communication mode with the host processor. Furthermore, it provides internal buffering of up to 40 characters and contains a 7 x 7 matrix character generator which accommodates 64 ASCII characters.

Industrial Digital Processor

Intel produces the iSBC 941 Industrial Digital Processor (IDP) which is programmed to handle an assortment of typical industrial digital interfaces and transducers. The controller can function to provide any of the following:

1. Scan up to 16 inputs for a change of state.
2. Provide up to 8 gated one-shot outputs.
3. Provide eight gated outputs with programmable pulse widths and periods.
4. Provide monitoring of up to 8 input lines for event sensing or as a programmable divider.
5. Provide the period measurement of up to eight inputs.
6. Provide a frequency to count conversion of one input.
7. Provide for the control of a stepper motor having up to eight phases.
8. Provide a simplex asynchronous serial input.

9. Provide a simplex asynchronous serial output.

In addition to providing one of the above functions, the IDP can also handle simple parallel I/O through the unused port inputs or outputs.

III. FUNCTIONS OF THE INTELLIGENT DIGITAL CONTROLLER

The iSBC 569 Intelligent Digital Controller (IDC) is a versatile digital I/O processor. The IDC is designed to operate in a system using any one of the following three modes:

1. Intelligent Slave
2. Stand-alone System
3. Limited Bus Master

Additional power is obtained by the utilization of three OBS's to generate up to 48 parallel input/output data lines.

In the intelligent slave mode, the controller's RAM is shared between the on-board 8085A and the Multibus users via a dual-port controller. Thus, a single bus master can control several intelligent slaves using the dual-port RAM as the major communications path. Switches are provided on the board to allow the user to reserve 1K bytes of RAM for use by the 569's processor only. This reserved memory is not accessible via the Multibus system interface and does not occupy any bus address space.

In the stand-alone mode, the entire system can consist of a single IDC, with cables, power supply and enclosure. An IDC can be installed at a remote site as a completely autonomous system.

The IDC may also be operated as a limited bus master when it is the only bus master in the system. Expansion memory and I/O boards may be connected to the IDC via the Multibus system bus to increase the input/output capabilities. This mode could be used to configure one IDC as a bus master with additional IDC's as intelligent slaves. This mode is not available with any other bus masters such as iSBC single board computers, disk controllers, or DMA devices.

Input/Output Functions

The I/O interface between the iSBC 569 Intelligent Digital Controller and the external devices to

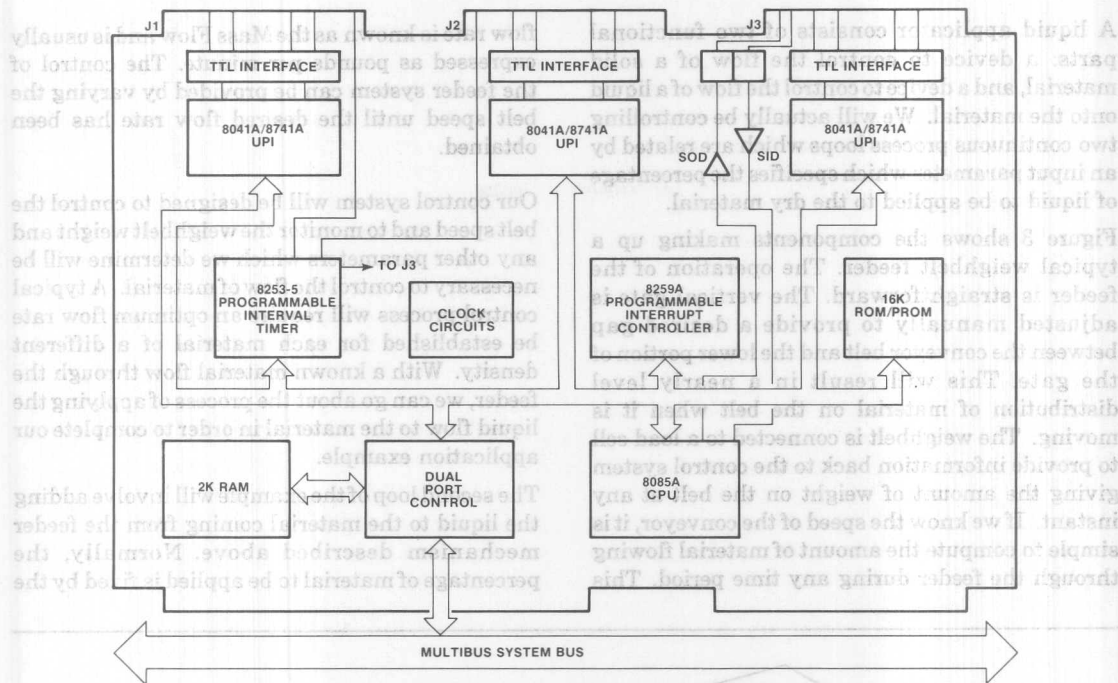


Figure 2. IDC Functional Block Diagram

which it is to be connected normally consists of various OBS devices. Each of these slaves has the ability to provide sixteen individual input and/or output lines. In addition, each provides two specialized input lines. The IDC is designed to accommodate up to three slave devices, so the normal I/O configuration of the board will consist of 48 digital data lines. If the specialized lines are considered, this number could be raised to 54. Sockets are provided for the insertion of drivers or terminators for use on the 48 digital lines. The 6 special purpose lines can only be used as inputs and are provided with pull-up resistors to terminate the input signals.

The driver/termination socket configuration limits the grouping of the I/O lines to be in groups of four. Any slave data line being used for an input must have its output latch placed into a logical 1 state so as to allow the input line to be controlled by the external signal.

IV. APPLICATION EXAMPLE

An example of the iSBC 569 controller in an application will help to explain the techniques used to implement a control system and to interface between the various functional units. The application chosen will consist of a typical use but will be simple enough to allow the design operations to be easily followed.

Suppose we choose to design a control system which will be produced as a subsystem to interface with and control a liquid applicator. As we go through the steps required to design and implement such a control system, we will see how the various hardware and software tools which are available from Intel can be utilized to allow easy implementation of the task.

Before proceeding, we will spend some time to insure there is a clear understanding about the definition of the liquid applicator. When this definition is complete, the design of the control subsystem can begin.

A liquid applicator consists of two functional parts: a device to control the flow of a solid material, and a device to control the flow of a liquid onto the material. We will actually be controlling two continuous process loops which are related by an input parameter which specifies the percentage of liquid to be applied to the dry material.

Figure 3 shows the components making up a typical weighbelt feeder. The operation of the feeder is straightforward. The vertical gate is adjusted manually to provide a desired gap between the conveyor belt and the lower portion of the gate. This will result in a nearly level distribution of material on the belt when it is moving. The weighbelt is connected to a load cell to provide information back to the control system giving the amount of weight on the belt at any instant. If we know the speed of the conveyor, it is simple to compute the amount of material flowing through the feeder during any time period. This

flow rate is known as the Mass Flow and is usually expressed as pounds per minute. The control of the feeder system can be provided by varying the belt speed until the desired flow rate has been obtained.

Our control system will be designed to control the belt speed and to monitor the weighbelt weight and any other parameters which we determine will be necessary to control the flow of material. A typical control process will require an optimum flow rate be established for each material of a different density. With a known material flow through the feeder, we can go about the process of applying the liquid flow to the material in order to complete our application example.

The second loop of the example will involve adding the liquid to the material coming from the feeder mechanism described above. Normally, the percentage of material to be applied is fixed by the

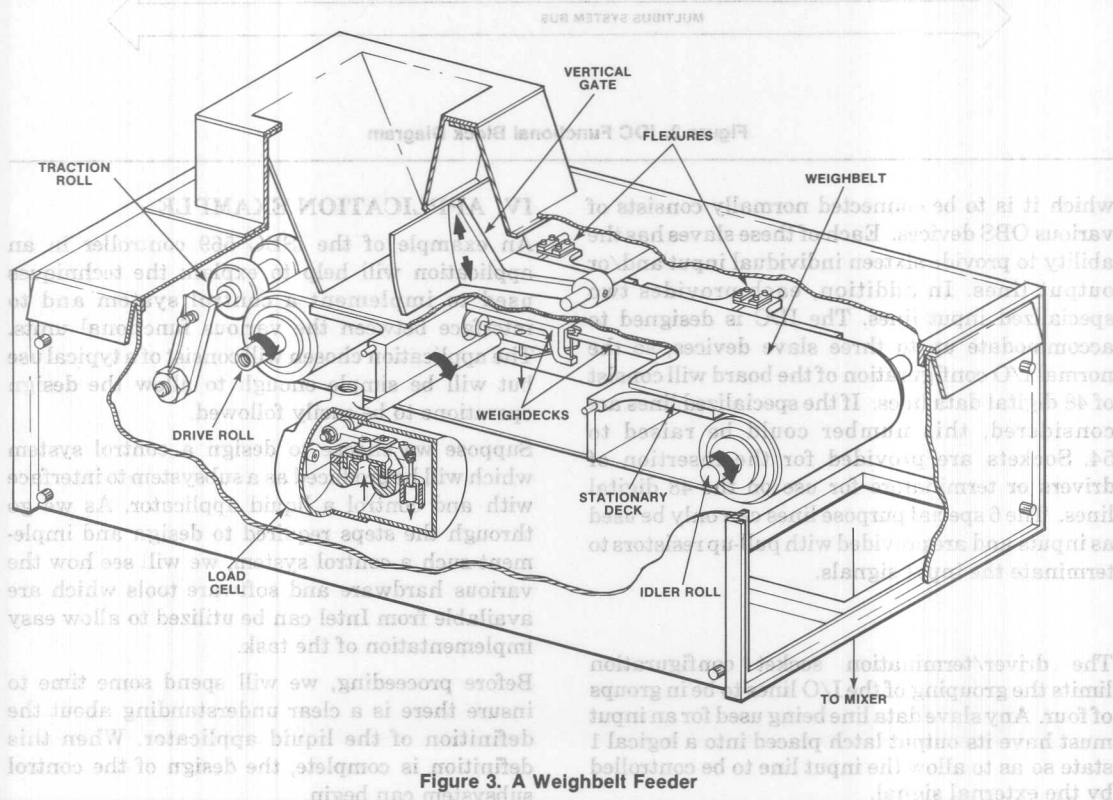


Figure 3. A Weighbelt Feeder

formula or mix design of the product which we are manufacturing. However, since the flow rate through the weighbelt feeder can and does vary (our first control loop will not always be able to exactly control the flow due to many conditions beyond our control), the liquid setpoint will constantly be changing as a function of the actual mass flow and the liquid percentage.

Figure 4 shows the liquid application piping diagram for the liquid portion of the control system. The items with which we will be directly concerned are the liquid flow meter and the control valve. The other components, while requiring consideration in an actual implementation, will be ignored in this application note for the sake of clarity. Let us consider the details of each control loop in more depth before we attempt to design the control system.

Mechanical Specifications

In subsequent portions involving development of the control system, we will be constantly referring to data regarding the mechanical specifications of the liquid applicator system. Therefore, we will

establish a set of theoretical technical specifications for our system. Later, we will see how close the control system can come to providing a control which meets or exceeds these parameters. These specifications will be broken down into two sets of data, one for physical parameters over which we have no control, and a second for the desired control characteristics.

The physical data provides information on the mechanical design and will be used for guidelines in selecting interface equipment and in preparing software algorithms. The physical data is:

Operating Belt Speed —

1.1 to 180 feet per minute. Adjusted by a variable speed motor directly coupled to the belt pulley mechanism.

Feed Output Rates —

Adjustable over a 10:1 range with a maximum output of 960 pounds per minute.

Feeder Belt Characteristics —

The belt will be 9 inches wide by 2 feet in length when installed. The belt pulley rollers will have a radius of 4.5 inches.

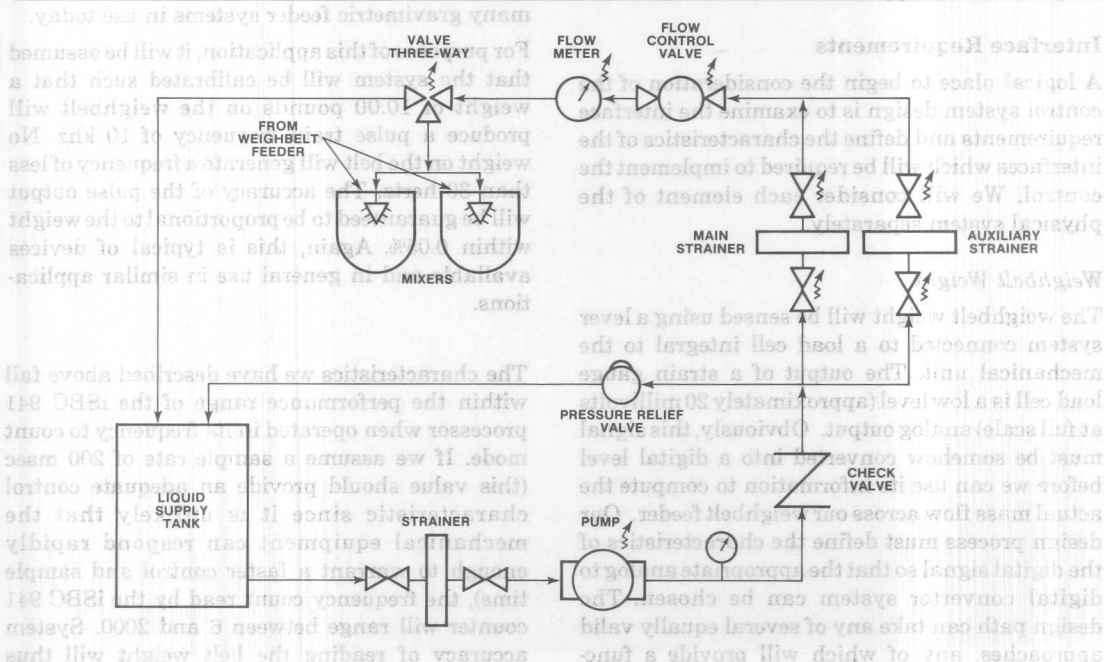


Figure 4. Liquid Flow Diagram

Feeder Weight Sensor —

The weighbelt feeder will incorporate a strain gauge load cell to measure the weight on the belt. Its linearity shall provide 0.1% of full scale range.

Liquid Flow Rates —

The liquid flow rates shall vary between 10.0 and 120.0 pounds per minute.

The desired operating characteristics of our control system will provide the following general responses:

Feeder Accuracy —

1% of full scale over a 10:1 range. The feeder will maintain the set feed rate within 1% of full scale over any one minute period. The minimum sample must be at least one pound.

Liquid Accuracy —

1% of full scale over the operating range. Must be able to track mass flow variations within the above limits.

These specifications will provide guidelines for the decisions which we will later make in providing a micro-computer control solution to the weighbelt feeder application.

Interface Requirements

A logical place to begin the consideration of the control system design is to examine the interface requirements and define the characteristics of the interfaces which will be required to implement the control. We will consider each element of the physical system separately.

Weighbelt Weight

The weighbelt weight will be sensed using a lever system connected to a load cell integral to the mechanical unit. The output of a strain gauge load cell is a low level (approximately 20 millivolts at full scale) analog output. Obviously, this signal must be somehow converted into a digital level before we can use its information to compute the actual mass flow across our weighbelt feeder. Our design process must define the characteristics of the digital signal so that the appropriate analog to digital converter system can be chosen. The design path can take any of several equally valid approaches, any of which will provide a functional control system. For the purposes of this

application note, we will assume that the design path will utilize the Intel iSBC 569 Intelligent Digital Processor.

This assumption requires us to utilize only signals which can be generated or interpreted using the computer board and its associated OBS's. We will not be capable of handling an analog signal. Since some type of signal conditioning would be required of the low level analog voltage anyway, this does not impose any serious restrictions on our design. Indeed, it will cause us to consider a technique which provides excellent noise rejection characteristics. We will assume that a voltage to frequency converter (V/F) will be installed near the load cell and the frequency will then be transmitted over a pair of wires to our digital interface. Commercially available converters provide a frequency output which varies between 0 and 10 kilohertz. With this in mind, we can continue with the development of the interfaces required in the application.

The load cell transducer will incorporate a local unit which generates a pulse train whose frequency is proportional to the weight upon the load cell. This mechanical arrangement is typical of many gravimetric feeder systems in use today.

For purposes of this application, it will be assumed that the system will be calibrated such that a weight of 10.00 pounds on the weighbelt will produce a pulse train frequency of 10 khz. No weight on the belt will generate a frequency of less than 30 hertz. The accuracy of the pulse output will be guaranteed to be proportional to the weight within 0.05%. Again, this is typical of devices available and in general use in similar applications.

The characteristics we have described above fall within the performance range of the iSBC 941 processor when operated in its frequency to count mode. If we assume a sample rate of 200 msec (this value should provide an adequate control characteristic since it is unlikely that the mechanical equipment can respond rapidly enough to warrant a faster control and sample time), the frequency count read by the iSBC 941 counter will range between 6 and 2000. System accuracy of reading the belt weight will thus exceed 0.1% of the full scale weight reading.

We will discuss the electrical and programming interfaces in subsequent sections of the application note.

Weighbelt Motor Control

The flow on the weighbelt will be controlled by changing the speed of the belt movement. Since the weighbelt is mechanically designed to maintain a constant bed level, the amount of material flowing will thus be adjusted.

The belt speed has traditionally been adjusted using either SCR controllers or by using variable transmissions between the motor and the conveyor belt. The increased utilization and development of stepper motors is leading toward greater use of direct stepper motor drives. This is the mode which will be utilized for this application.

The manufacturer's specifications for the weighbelt indicate that the following requirements exist for driving the device:

REQUIRED TORQUE — 149 LB-IN-IN

REQUIRED MAX SPEED — 2.54 REV/SEC.

Referring to typical manufacturer specification sheets for stepper motors, we find the torque vs. speed characteristics shown in Figure 5. Our application requires 2.54 revolutions/sec which translates to 508 steps per second when the stepper is used in a 1.8 degree per step mode. We can see that the requirements fall well within the capabilities of the particular motor.

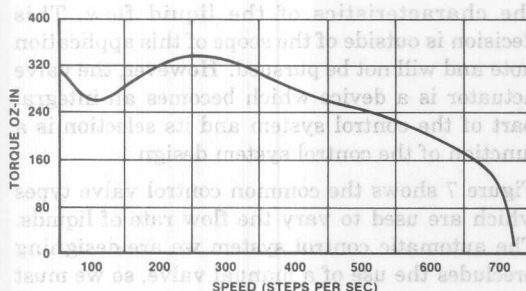


Figure 5. Stepper Motor Torque/Speed

At this point, we have four routes which may be pursued to actually interface with the motor. These are:

1. Utilize the iSBC 941 stepper mode to drive the stepper motor directly.
2. Utilize the iSBC 941 frequency generation mode to drive a standard stepper translator.
3. Utilize parallel outputs to provide a digital output to a stepper translator.
4. Utilize a 4-20 ma. current signal to a stepper translator.

Three of the above modes use a translator to drive the motor. If possible, we should strive to eliminate the cost of this intermediate device.

Again, we will refer to the published motor specification sheets. For our typical motor, the data is shown in Figure 6. The requirement for providing in excess of six amperes per winding exceeds the capabilities of the output drivers which can be installed on the iCS 930 termination board. We will be forced to either design a custom high power driver board or to use a translator module. To keep the application as simple as possible, we will choose the latter.

ELECTRICAL RATINGS 1.8 DEGREE STEPPING MOTOR

Motor Type	Time for One Step	DC Volts	Amperes Per Winding	Resistance Ohms	Inductance Millihenries
Ourtype	1.7 msec	2.3	6.1	0.37	2.4

Figure 6. Stepper Electrical Ratings

We have three choices left when the decision has been made to use a translator module. The use of a current output mode will necessitate the use of an external analog board. This is undesirable, both from the standpoint of interboard communication requirements, and from a cost effective basis.

The use of a parallel output would commit many of our output data ports and would require the installation of UPI modules or iSBC 941 modules to get the parallel output drivers. In addition, parallel digital input is not a common option of commercially available translators.

This leaves us with the use of a variable frequency output to provide stepping information to the translator module. This is a normal operational mode of the iSBC 941 processor and the required 508 hertz is within the normal output range of the device.

A definite advantage of our decision to use a stepper motor drive for the weighbelt is that we do not have to maintain accurate feedback and control algorithms to maintain the conveyor speed. Only a simple check need be made to verify that the conveyor has not stalled. The stepper motor will inherently maintain a speed proportional to the frequency rate.

The actual electrical and programming interfaces will be discussed in subsequent sections of this application note.

Weighbelt Speed Measurement

We have mentioned that a control system using a stepper motor for speed control can operate effectively in an open loop configuration. However, since a faulty component could result in failure of the motor to run, we must verify that the belt is indeed moving. This is easily accomplished by adding a magnetic sensor to the weighbelt rollers and counting the pulses generated as the device operates.

Typical magnetic sensors and ring magnets for installation on the weighbelt will provide us with ten pulses per revolution of a belt pulley. Since the pulley is operating at a maximum speed of 2.54 revolutions per second, we will receive between 0 and 25.4 counts per second. Using our sample period of 200 milliseconds, this means that we will count between 0 and 5 counts during each time interval. Our decision to use a stepper control loop rather than a conventional closed loop seems justified as we would obtain rather poor control with feedback having this poor of resolution.

We must make a decision to determine how the speed will be sensed by the control board. An obvious choice would be the use of an iSBC 941 processor operating in the period measurement mode. This would require using our third socket on the iSBC 569 host board and would leave us without the ability to use an additional device to support the liquid control loop. We should seek an alternative solution.

The iSBC 569 controller board provides an 8253 programmable interval timer. A first approach might be to attempt to configure one of these counters to provide an event counting mode and read the belt speed from the counter. However, this is not possible since we would be required to zero the counter after each reading and the counter does not load the preset count until a clock pulse is present. We would have no ability to distinguish between no belt motion and the belt motion which is the same as the previous reading!

An alternative approach is to create a software counter by routing the belt movement pulse to one of our interrupts and creating a program which will increment a counter. Each time a count is sensed, the software will increment a memory location by an increment which corresponds to the speed represented by one count.

Again, we will delay the discussion of the electrical and programming interfaces until subsequent sections of this application note.

Liquid Flow Control

The design of a control system to provide control of flow through a liquid valve is an integral part of the liquid pipe and plumbing design. To optimize the system operation and provide a system at the minimum cost, the integration of control and mechanical design must be made.

Several possibilities exist when making a decision as to which control valve to use in adjusting the liquid flow rate. The actual selection of the physical valve mechanism should be based upon the characteristics of the liquid flow. This decision is outside of the scope of this application note and will not be pursued. However, the valve actuator is a device which becomes an integral part of the control system and its selection is a function of the control system design.

Figure 7 shows the common control valve types which are used to vary the flow rate of liquids. The automatic control system we are designing precludes the use of a manual valve, so we must make our selection between the air actuated and the motorized control valve.

Classical control design has utilized air actuated valves almost exclusively. This type of actuator incorporates an intermediate transducer to

PROPORTIONAL CONTROL VALVES

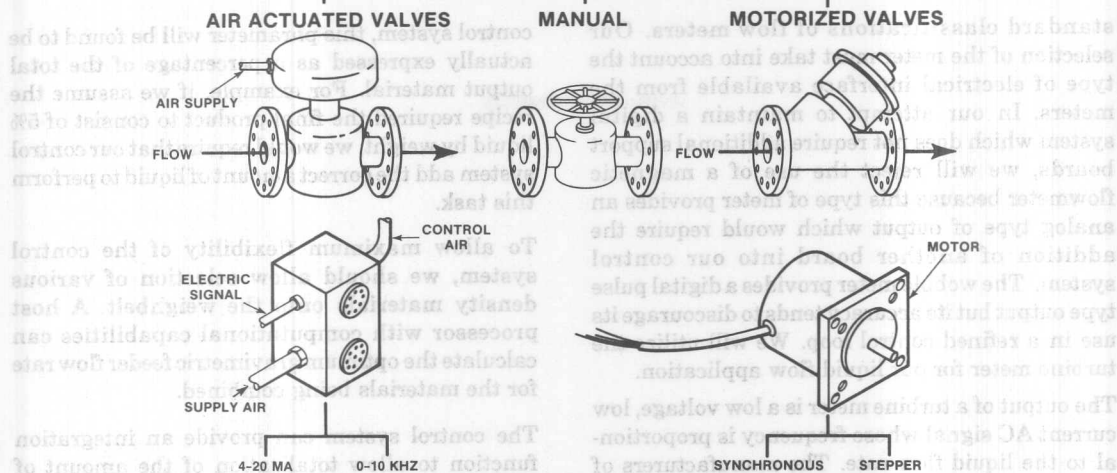


Figure 7. Control Valve Family

convert the signal generated by the control system into a variable air pressure. This air is used to drive a pneumatic control actuator. Two types of electrical to pneumatic transducers are in common use. The most prevalent converts a 4 to 20 milliamperes control signal into a proportional air signal. The second type will accept a 0 to 10 kHz pulse train and convert this to an air output.

Both of the above systems provide excellent electrical noise immunity and give reliable operation in industrial environments. They do, however, have disadvantages. A supply of air must be present at the control devices and this air must be maintained such that it is free from water and oil. In many cases, this presents costly installation and maintenance considerations. The use of computerized control systems has led to a recent concept of eliminating the intermediate conversion and using instead a digitally controlled actuator.

A stepper motor can be connected to the actuator

of the control valve to provide a simple and economical control path. The control outputs from the PID control loop can be sent to the iSBC 941 processor's command queue and the controller will handle the motor movements.

The electrical and programming interfaces of this interface will be fully discussed in subsequent sections.

Liquid Flow Measurement

The use of a liquid control valve to vary the liquid flow cannot in itself provide an accurate control loop. Because the flow rate through a fixed valve will vary with material densities, temperatures, and pressures, we must provide some type of feedback into our control algorithm. Thus, a flowmeter must be inserted into the liquid flow and its output returned to the system.

The control system designer can choose from several types of flow meters depending upon his requirements. Figure 8 shows many of the more

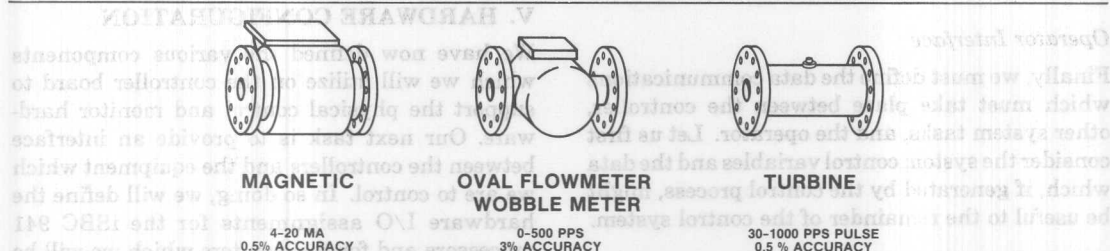


Figure 8. Flow Meter Classifications

standard classifications of flow meters. Our selection of the meter must take into account the type of electrical interface available from the meters. In our attempt to maintain a digital system which does not require additional support boards, we will reject the use of a magnetic flowmeter because this type of meter provides an analog type of output which would require the addition of another board into our control system. The wobble meter provides a digital pulse type output but its accuracy tends to discourage its use in a refined control loop. We will utilize the turbine meter for our liquid flow application.

The output of a turbine meter is a low voltage, low current AC signal whose frequency is proportional to the liquid flow rate. The manufacturers of the meters provide pre-amplifiers which convert the signal into 10 volt peak to peak square waves which are equivalent in frequency to the AC pulses. The operating frequency ranges typically from 100 to 1200 pulses per second.

It is desirable to measure the flow rate using a single iSBC 569 controller. If we consider that a 200 millisecond control interval will be used, the flow will result in a reading of between 20 and 240 pulses per sample period. These readings could be performed using an iSBC 941 processor, but we do not have the socket available for a fourth module, so we must consider utilizing another interrupt driven software counter as was done with the belt speed.

All control and monitoring equipment for our liquid control application has now been defined in such a manner as to be compatible with the utilization of a single iSBC 569 controller board. The actual interfaces to perform the interconnections and to provide control instructions can soon be considered.

Operator Interface

Finally, we must define the data communications which must take place between the controller, other system tasks, and the operator. Let us first consider the system control variables and the data which, if generated by the control process, might be useful to the remainder of the control system.

The first variable which comes to mind is the liquid flow setpoint. If we consider the entire

control system, this parameter will be found to be actually expressed as a percentage of the total output material. For example, if we assume the recipe required the final product to consist of 5% liquid by weight, we would require that our control system add the correct amount of liquid to perform this task.

To allow maximum flexibility of the control system, we should allow selection of various density materials onto the weighbelt. A host processor with computational capabilities can calculate the optimum gravimetric feeder flow rate for the materials being combined.

The control system can provide an integration function to allow totalization of the amount of material which has been transferred through the system. A capability of outputting the amount of material which has passed over the weighbelt and the amount of liquid added will be included.

The implications of the parameter storage and generation will be dealt with later when the host/slave relationships of the iSBC 569 controller are discussed.

Interface Summary

We have defined the required interfaces which will be needed to perform our control task. These can be grouped into external and internal interfaces. The external interfaces are those which connect to physical pieces of external equipment.

These are summarized in Figure 9. The internal interface relates to the data which is to be passed between the iSBC 569 Intelligent Slave board and other boards which may be present on the MULTIBUS system bus. These data areas are shown in Figure 10.

V. HARDWARE CONFIGURATION

We have now defined the various components which we will utilize on the controller board to support the physical control and monitor hardware. Our next task is to provide an interface between the controllers and the equipment which we are to control. In so doing, we will define the hardware I/O assignments for the iSBC 941 processors and for the counters which we will be utilizing. The following paragraphs will deal with the optimization of this configuration.

**** DEVICE ****	**** SIGNAL TYPE ****	**** BOARD ELEMENT ****
WEIGHBELT MOTOR	10 VDC PULSE	ISBC 941
WEIGHBELT WEIGHT	10 VDC PULSE	ISBC 941
WEIGHBELT SPEED	110 VAC PULSE	8259A INTERRUPT
LIQUID VALVE	5 VDC MULTIPHASE	ISBC 941
LIQUID FLOW	10 VDC PULSE	8259A INTERRUPT

Figure 9. Control/Monitor Signals

*** INPUTS ***	*** OUTPUTS ***
GRAVIMETRIC FLOW	ACCUMULATED SOLIDS
LIQUID PERCENTAGE	ACCUMULATED LIQUID

Figure 10. Communication Signals

Controller Interface

Good design practice dictates that we should provide optical isolation between the controller and the external equipment when designing for an industrial environment. The optical isolation is included if we utilize the Intel iCS series of signal conditioning/termination boards. We find that we have two types of digital termination panels available, one for low current, low voltage applications and second for higher current and voltage uses. If we base our choice on the data provided by Figure 8, we will lean toward using the iCS 930 panel for our interface. This board can handle a mixture of signal levels and will support up to sixteen individual lines, providing almost double our needs.

Even a cursory glance at the ISBC 569 controller will provide the knowledge that three edge connectors are utilized to bring the OBS signals from the board. This would indicate that the simplest (and most costly) solution is to use three termination panels. Obviously, we should investigate further before making such a decision. Three possibilities are readily apparent. First, we might

perform some type of re-routing of data lines on the board so as to use only one connector. Second, we can use more than one connector on the ribbon cable and perform a parallel connection of the various lines and choose them so that no duplication of lines results. Finally, we can use some scheme of connecting three cables to the board and use the optional Port C connectors on the termination panel.

The schematic drawings of the IDC indicate that only six of the OBS I/O lines of each processor socket are broken by wire wrap jumper posts. All of the lines so configured are on the Port 2 data lines. Unless we decide to cut etch and add soldered wires, we will not be able to configure our board with this technique. Some further investigation is in order before we can make a decision. The use of a parallel output technique using multiple connectors on a single cable seems to present a feasible approach if we can work out an assignment of I/O which will not cause conflicts. We will begin by building a trial port assignment table in which we will assign the required functions to input/output ports. We will group the inputs and outputs into groups of four to handle the terminator/driver arrangement which is built into the board. This table is shown in Figure 11. We obviously have a small problem. We have

Port	Socket 1	Socket 2	Socket 3	Direction
10		Weight In		In
11				In
12				In
13				In
14				
15				
16				
17				
20	Conv. Mtr.			Out
21				Out
22				Out
23				Out
24			Valve Ph. 1	Out
25			Valve Ph. 2	Out
26			Valve Ph. 3	Out
27			Valve Ph. 4	Out

Figure 11. UPI™ Socket to Terminator Initial Assignments

not yet shown the signals from the conveyor speed and the liquid flow into the on-board interrupt counters. The schematics show that these signals are brought onto the board on the edge connectors but the locations correspond to Port C lines which do not exist on the iCS 930! We have available input lines on the Port 1 connectors but there is no provision to break the signal on the board to route it to the counter interrupts.

If we move on to the third alternative, we find that the interconnection paths caused by tying various lines together cause even greater problems. Either some fact must have been overlooked, or we must consider the use of more than

one terminator board.

Figure 11 indicates that three lines are available on the Port 2 data lines which go to jumper posts and which could be used if they were not part of an output driver of Port 20. If some technique can be found to use these "output" lines as inputs, our problem will be solved. The use of an open collector driver can provide us with the ability to use the line as an input so long as the drivers are turned off. This should be no problem as we can force the outputs to this state either through the appropriate jumpering of inputs or by outputting data to the OBS 1 ports corresponding to these bits. The resulting electrical configuration can be seen in Figure 12.

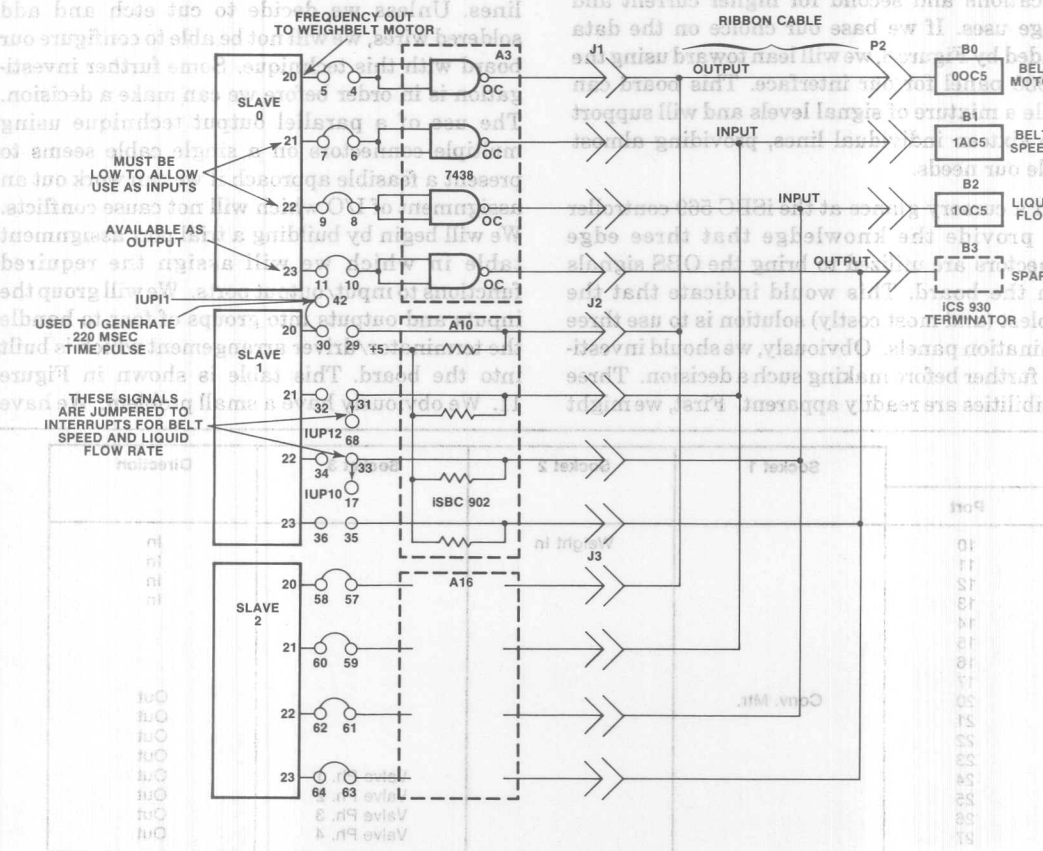


Figure 12. Port Assignments 20-23

Let us examine the implications of performing this interconnection. The physical layout of the board and the use of the terminator/driver sockets causes the I/O lines to be grouped into sets of four data lines. We must choose which of the three iSBC 941 modules will be responsible for supporting each of the lines. In Figure 12, we can see that the belt motor is driven by OBS Socket 1, Bit 20. This requirement has placed output drivers onto data Bits 21, 22 and 23. Our requirement is to provide two signals which can be routed to the counter inputs so we must place a terminator into either socket A10 or A16. We have arbitrarily chosen to use socket A10. The use of the terminators in parallel with the drivers will not create a problem so long as those lines which are used as inputs

have the driver in the high impedance state. This is done by requiring that the output Bits 21 and 22 of the device placed into socket 1 are driven low. Finally, we see that the remaining Bit 23 may be used as a general purpose output line if it becomes required.

The wiring configurations for the remaining connector groupings are shown in Figures 13, 14 and 15. In Figure 13, we see the assignments which can be used for Bits 10, 11, 12 and 13. We have earlier defined that an iSBC 941 processor would be used in a high speed frequency counting mode to determine the weighbelt weight. This device will be placed into socket 2. The use of this mode precludes the use of any general purpose

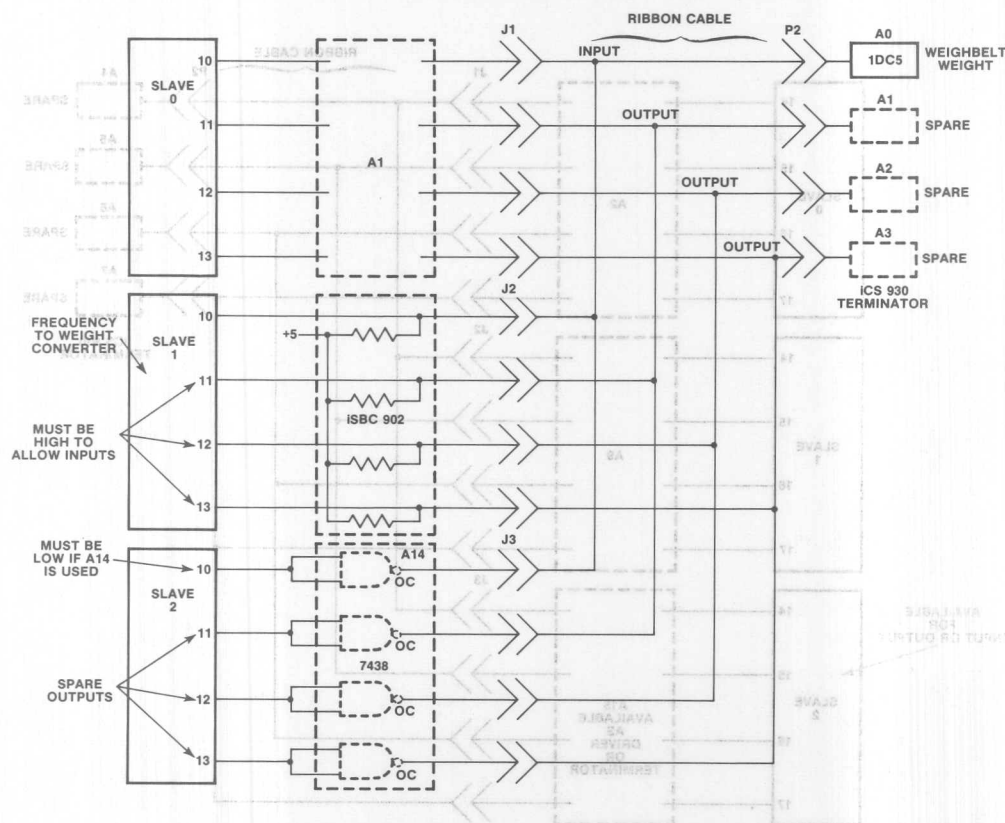


Figure 13. Port Assignments 10-13

input/output operations of the processor if we desire to maintain maximum accuracy of the frequency measurement. We will arbitrarily choose to use Bit 10 as the location of the frequency count input. This will necessitate installing a terminator into the socket corresponding to the processor input. If required, we can install open collector drivers into socket A14 and use the remaining three bits for general purpose outputs. If this is done, care must be taken to assure that Bit 10 of the device which is placed into socket 3 is placed into a low state as was done in the preceding example.

The interconnection scheme for Ports 14 through 17 can be seen in Figure 14. Note that no ports of this group are dedicated to our defined control

functions. These four bits may be used as inputs or outputs as required by the application. For example, we have ignored the fact that actual control loops incorporate solenoids for flow control routing. The unused bits can be used to perform these tasks.

Figure 15 shows the interconnections for the remaining group of bits. There are several features shown on this drawing which should be discussed in some detail. Let us first consider the remaining function which we must implement. This is the control for the liquid valve stepper motor. An iSBC 941 IDP operating in the stepper mode will provide the necessary control functions to drive the motor. Since all four of this group's

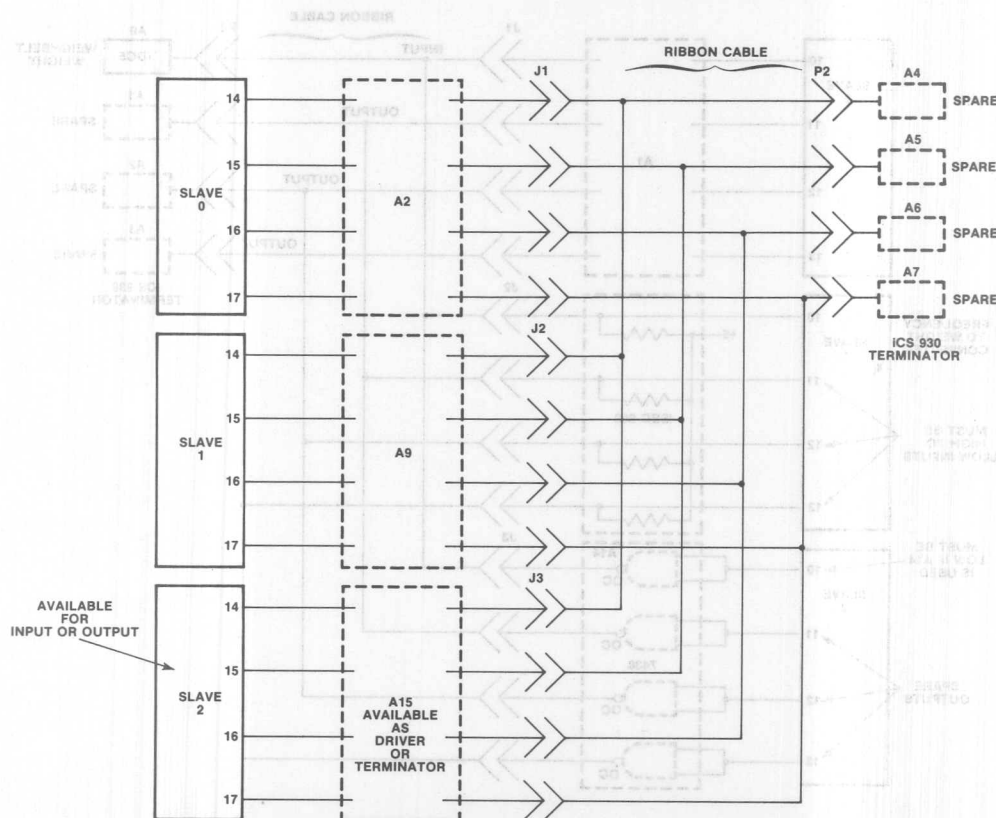


Figure 14. Port Assignments 14-17

data lines are committed to drive the four phases of the stepper motor, there are no other functions available.

An important feature of the iSBC 941 processor is illustrated in Figure 12. This is the ability to enable the processor to generate an interrupt at some point in its operation. We have earlier indicated that we will use the processor in socket 2 (the frequency counter) to provide us with a 200 msec time reference. When the iSBC 941 processor is enabled with an ENFLAG command and is operating in the frequency count mode, it will generate an interrupt on its output line, Port 25. Figure 15 shows how this interrupt can be connected to the host board's internal interrupt input structures.

The hardware configuration has been defined through Figure 14. The actual implementation can be handled through the use of the various wire-wrap jumpers on the IDC. Drivers and terminators can be installed as indicated in the preceding discussion.

VI. SOFTWARE CONFIGURATION

As with most computer controlled systems, the actual implementation of the task is handled with software. In older designs and in many mini-computer systems, this task has become formidable and has resulted in cost over-runs and schedule delays. Intel provides many tools for use by the designer to prevent this type of problem and to assist him in easily creating a workable and well

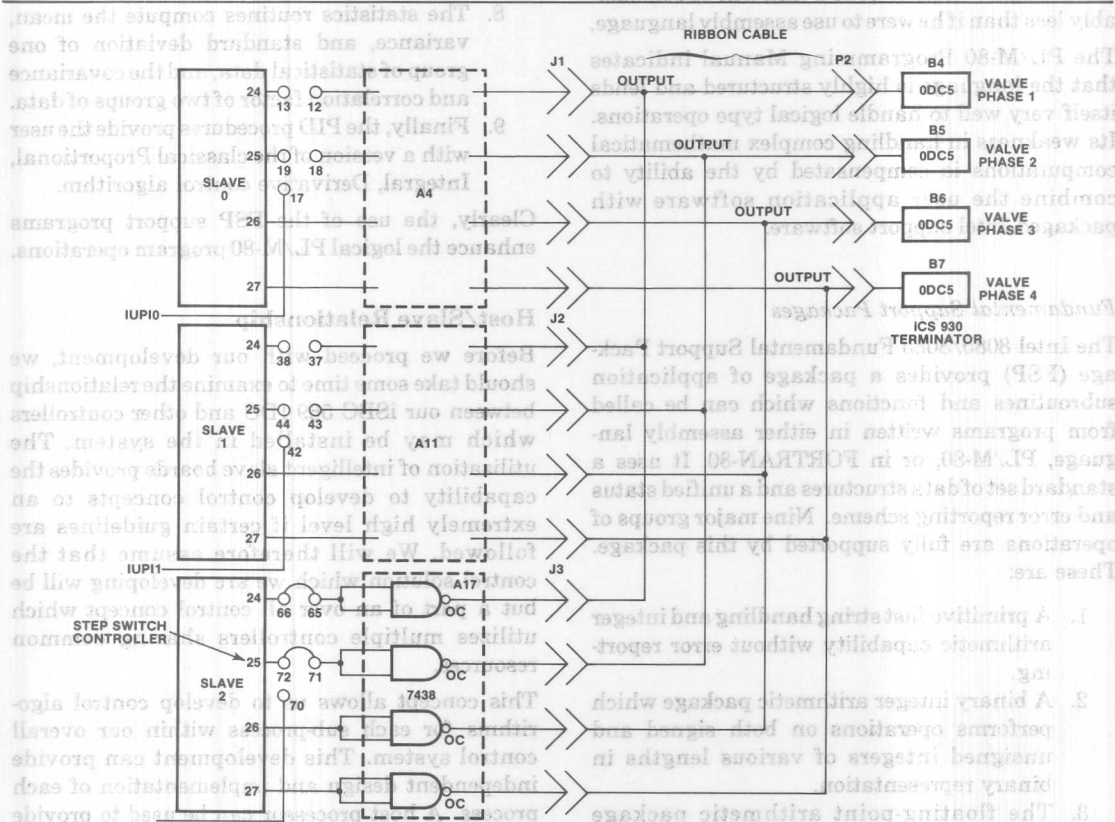


Figure 15. Port Assignments 24-27

documented software configuration. Let us look at some of these tools in more detail and consider how their use will help us to write our programs easily and quickly.

High Level Programming Languages

A valuable tool, which Intel provides the designer of small control systems, is the ability to program even the smallest systems using a high level programming language, PL/M-80. This language offers relatively efficient and structured, programming capabilities. It has been determined that PL/M-80 users can expect to use between 1.1 to slightly more than 2 times as much program memory as would be used for the same task written in assembly language. At the same time, the programmer's time to code a task will be considerably less than if he were to use assembly language.

The PL/M-80 Programming Manual indicates that the language is highly structured and lends itself very well to handle logical type operations. Its weakness in handling complex mathematical computations is compensated by the ability to combine the user application software with packaged Intel support software.

Fundamental Support Packages

The Intel 8080/8085 Fundamental Support Package (FSP) provides a package of application subroutines and functions which can be called from programs written in either assembly language, PL/M-80, or in FORTRAN-80. It uses a standard set of data structures and a unified status and error reporting scheme. Nine major groups of operations are fully supported by this package. These are:

1. A primitive fast string handling and integer arithmetic capability without error reporting.
2. A binary integer arithmetic package which performs operations on both signed and unsigned integers of various lengths in binary representation.
3. The floating-point arithmetic package which provides operations on floating point numbers in four formats: single precision, single-precision extended, double precision, and double-precision extended.

4. The decimal arithmetic routines which perform integer and fixed point computations on numbers which are stored as strings of ASCII characters.
5. A string handling section which contains routines to transform strings and to extract and insert substrings. A routine for scanning of general input and one for formatting of general output are included.
6. Routines for number conversion, for numeric I/O transformation of data from one format to another, input scanning of numeric strings, and formatting of numeric strings for output are also available.
7. The floating point transcendental function section provides trigonometric, exponential, and other transcendental functions.
8. The statistics routines compute the mean, variance, and standard deviation of one group of statistical data, and the covariance and correlation factor of two groups of data.
9. Finally, the PID procedures provide the user with a version of the classical Proportional, Integral, Derivative control algorithm.

Clearly, the use of the FSP support programs enhance the logical PL/M-80 program operations.

Host/Slave Relationship

Before we proceed with our development, we should take some time to examine the relationship between our iSBC 569 IDC and other controllers which may be installed in the system. The utilization of intelligent slave boards provides the capability to develop control concepts to an extremely high level if certain guidelines are followed. We will therefore assume that the control solution which we are developing will be but a part of an over all control concept which utilizes multiple controllers sharing common resources.

This concept allows us to develop control algorithms for each sub-process within our overall control system. This development can provide independent design and implementation of each process. A host processor can be used to provide any required inter-process communication tasks and to provide the operator interface. We have previously indicated that the operator interface will provide some means to adjust the weighbelt

feeder setpoints and the liquid ratio. It should also allow the operator to display the current status of the process. Since these operator interface functions are but a part of the overall control functions, the interface should be programmed such that maximum flexibility can be gained through its use. Fortunately, such an interface is available using Intel's RMX/80 BASIC-80.

RMX/80 BASIC-80 Interpreter

The RMX/80 BASIC-80 Interpreter is a high level language interpreter with extended disk capabilities. It operates on iSBC 80 Single Board Computers and allows the interpretation of BASIC-80 source code into an internally executable form. Many other features are available and many configurations are possible depending upon the exact system requirements (refer to the *BASIC-80 Reference Manual*, 9800758).

Maximum utilization of the operator interface with a minimum of development time can be achieved with the preconfigured version of the software/hardware package. This will provide us with complete disk I/O capabilities and the ability to easily program and maintain any programs which may become necessary to implement the interface. The actual implementation of the interface will be done later, after we have defined the control task.

Software Tasks

The task of preparing the software can be broken down into three major groupings or tasks. These are defined to be:

Prepare the Software Drivers.

This involves defining the relationships between the control algorithm parameters and the input/output hardware devices and creating software to implement these definitions.

Prepare the Control Algorithm.

This will involve developing a control algorithm which defines the relationships between the various system parameters. This

algorithm will draw heavily upon the resources of the FSP programs and the software drivers which relate the parameters to the physical hardware.

Finally, the operator interface must be defined which will relate the parameters used in the control scheme to other controllers and to the operator. This will allow the control task to interact in such a manner as to provide a meaningful element of the overall control concept.

VII. SOFTWARE DRIVERS

Before developing the actual control algorithm, we must create the drivers which communicate with the three iSBC 941 processors in their assigned operating modes. We will define two driver sections for each processor, one to handle the initialization, and a second to provide the ongoing communications as required by the control algorithm program.

Motor Speed Control Processor

The first processor which we will discuss is to be located in slave socket number 0 and will be used to produce a variable frequency output. Let us consider in some detail how this can be accomplished using an iSBC 941 Processor. First, consider the task of initializing the device to the primary function operating mode, FREQ.

Referring to the *iSBC 941 Industrial Digital Processor User's Guide*, we find that the initialization requires the sequence of commands and data shown in Figure 16. We will identify the meaning of each of these terms and create a software

Description	Command/Data
Request INIT	C
FREQ Select	D
Scale Factor	D
Output Enable	D
Initial State	D
P20 Delay	D
P20 Period	D
Request PAUSE	C

Figure 16. FREQ Initialization

program which will handle the required initialization of the processor. The purpose and use of the various commands to the processor are well defined in the user's guide and will not be repeated here.

The first byte of data, which must be sent following the initialization command, is the data byte signifying that the operational mode is to be the frequency output. This is defined in the manual as being equal to the data byte "0B5H" or "035H" as expressed in the hexadecimal numbering system. The choice of values to be sent is dependent upon our desire to utilize the internal or external time reference period for the operations. If we utilize the internal time reference, our minimum increment or resolution of operations will be 86.72 microseconds.

To determine if this speed is adequate for our frequency generator, we must consider the impact that this resolution has on the output. A 550 hertz signal has a period of 1.82 milliseconds. If we increase this period by the 86.72 microsecond time reference, we find that the next increment in the frequency generators output will be approximately 372 hertz. This resolution is certainly not adequate to meet the motor control requirements! We should consider using the external clock to provide the time reference. One of the 8253 Interval Timers on the iSBC 569 board can be used to generate a reference time. If we arbitrarily choose to use a 10 microsecond reference to the IDP, we find that the worst case resolution for the 550 hertz signal becomes about 4 hertz. This is certainly within our requirements of motor control. The primary function signal should then be sent as a "0B5H".

The second byte is used to establish a scale factor for the processor. This scale factor is used to generate the basic time increment which can be used to establish the frequency output; that is, the minimum time increment which can be used to establish a period or pulse width will be the scale factor times the reference time period.

In our case, because of the wide frequency output range, we cannot specify the scale factor at initialization (later data will show the need for

multiple scale factor ranges). We will then only need to send some arbitrary value at initialization to allow the processor to complete its initialization sequence.

The Output Enable data byte is used to select which of the Port 2 output bits are to be used to generate the output signals. The hardware configuration established earlier placed the output onto Bit 0 of the port, so this data byte shall be specified as a byte having only Bit 0 set to a logical one or equal to 01H.

The Initial Output parameter specifies whether each bit selected as an output by the output enable byte is to be initially set to a logical one or zero when the processor is first enabled. For this application, it really does not matter, but we will arbitrarily pick the state to be equal to zero. The byte will be defined as being set to 00H.

The Delay parameter is used to define the waveform which will be generated and specifies the number of time increments which must elapse before the waveform will change states. Rather than to constantly vary the delay to maintain a square wave output, we can choose an arbitrary value of one time increment before changing state. The output will have a varying duty cycle as the frequency changes. This should cause no problems for the translator driving the weighbelt motor. The byte will be defined as being set to a value of 01H.

Finally, the Period of the waveform must be chosen. Again, this parameter will be changed according to the desired frequency, so only an arbitrary value need be sent. Indeed, since this is the last parameter, the value could be omitted entirely by sending the PAUSE command in its place.

The initial data definition can be defined using PL/M-80 language conventions as a block of six bytes as shown in Figure 17.

The actual communications between the host processor on the iSBC 569 board and the IDP utilizes the protocol explained in previous sections of this note. The status register of the IDP will be tested for the bit signifying that the input buffer

```

22 1 /* DECLARATION OF ISBC 941 #0 INITIALIZATION DATA */
23 1 DECLARE FREQ LITERALLY '0B5H';
24 1 DECLARE SF LITERALLY '000H';
25 1 DECLARE OUTPUT$ENABLE LITERALLY '00101H';
26 1 DECLARE INITIAL$STATE LITERALLY '000H';
27 1 DECLARE DELAY LITERALLY '001H';
27 1 DECLARE PERIOD LITERALLY '000H';

34 1 /* DECLARATION OF ISBC 941 PRIMARY DATA */
34 1 DECLARE INIT$TABLE(6) BYTE DATA (
    FREQ,
    SF,
    OUTPUT$ENABLE,
    INITIAL$STATE,
    DELAY,
    PERIOD );

```

Figure 17. Initial FREQ Data Field

full is not set. This will indicate that the device is ready to accept either a command or a data byte. The command to request a primary function will be sent. At this point, the processor will be expecting a series of data bytes as specified by the

function being selected. A "Do Loop" can be used to effectively transmit this data to the device. The program to perform this function is illustrated in Figure 18.

```

44 2 /* REQUEST PRIMARY FUNCTION */
45 2 DO WHILE ( (INPUT (UPI$0$STATUS) AND IBF) < > 0);
46 2 END;
46 2 OUTPUT (UPI$0$COMMAND) = INITPF;

47 2 /* LOAD INITIAL PARAMETERS */
48 2 DO I=0 TO 5;
49 3 DO WHILE ( (INPUT (UPI$0$STATUS) AND IBF) < > 0);
50 3 END;
51 3 OUTPUT (UPI$0$DATA)=INIT$0$TABLE(I);
51 3 END;

52 2 /* TERMINATE PARAMETER LOADING */
53 2 DO WHILE ( (INPUT (UPI$0$STATUS) AND IBF) < > 0);
54 2 END;
54 2 OUTPUT (UPI$0$COMMAND)=PAUSE;

55 2 /* START FREQUENCY FUNCTION */
56 2 DO WHILE ( (INPUT UPI$0$STATUS) AND IBF) < > 0);
57 2 END;
57 2 OUTPUT (UPI$0$COMMAND)=LOOP;

```

Figure 18. IDP Initialization

When all required data parameters have been sent, the data portion of the initialization is terminated by sending a PAUSE command as shown in Figure 18. Note how, in each case before data or a command is sent, we wait until the input buffer is empty. Finally, the initialization is completed when we have sent the LOOP command. The processor will now be generating an output frequency as specified by the parameters.

Remember that, according to our earlier discussion and as we have shown in Figure 12, the unused output ports should be set to a logical low condition to allow the use of those lines as inputs to carry additional data into the controller. This should be done as a part of the initialization process. The secondary utility command, CLRP2 is used for this purpose. This process is illustrated in Figure 19.

We should next direct our attention to establishing a software interface which will take the desired

weighbelt speed term and convert it to a frequency output suitable to drive the motor translator. We know that this will involve selecting a particular scale factor and period term which will generate the correct waveform. Previously, we established that, for a maximum frequency of 550 hertz, we need to establish a period of 1.82 milliseconds. Many combinations of Scale Factor and Period parameter will generate this time interval. Ideally, the smallest increment of change can be established by setting a constant period and modifying the scale factor. If we make some calculations, we will find that the fact that the scale factor is a byte value (giving us a range of between 0 and 255) limits the frequency range which can be produced using any one value for a period. It seems that we will be forced to vary both the period and the scale factor as a function of the desired frequency.

In Figure 20, we have plotted the frequency output for various values of Scale Factor and Period. Our

```

/*SET UNUSED BITS TO ALLOW EXPANSION */
59 2 DO WHILE ((INPUT UPI$STATUS) AND IBF) < > 0;
59 3 END;
60 2 OUTPUT (UPI$COMMAND)=CLRP2;

61 2 DO WHILE ((INPUT (UPI$STATUS) AND IBF) < > 0;
62 3 END
63 2 OUTPUT (UPI$DATA)=INITIAL$OUTPUT;

```

Figure 19. Secondary Utility Command

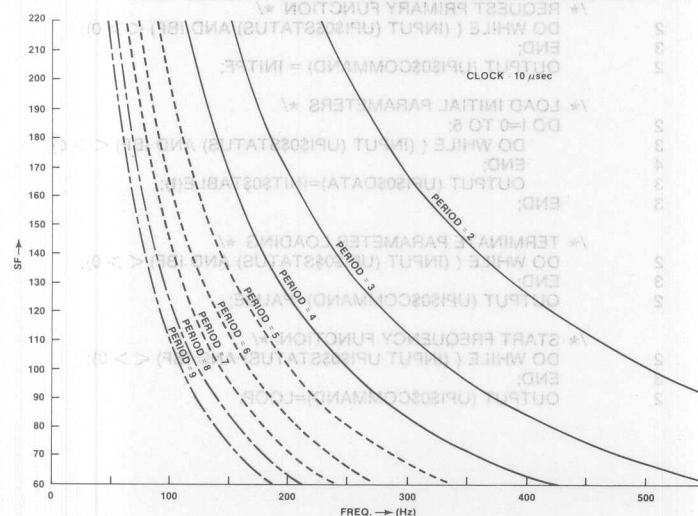


Figure 20. Frequency Vs. Parameters

intent is to maintain the highest resolution possible for the desired output range of 50 to 550 hertz. Choosing four period base parameters will provide us with acceptable waveform generation characteristics. We will choose the data sets of Figure 21 based upon the data shown in Figure 20.

The Period can be determined by examining the desired frequency range. The scale factor can be calculated from the equation:

$$SF = 10,000 / ((FREQUENCY) \times (PERIOD))$$

Again, the PL/M-80 language program to implement the interface between the host and the IDP is easily constructed. For example, Figure 22 provides the code which will be required to determine the appropriate Period parameter and also illustrates the use of FSP programs to handle

the mathematical calculations required to determine the corresponding scale factor.

The principles above can be expanded into a complete interface package to offload the host processor of the need to generate the frequency waveform to the translator of the weighbelt motor. The complete program for the processor can be found in Appendix A.

Weight Input Processor

The second use of an iSBC 941 Processor is to provide the capability of converting the high frequency inputs from the weight sensor of the weighbelt into a digital value equivalent to the actual weight on the belt. This frequency to digital conversion can be easily accomplished by the use of the Primary Function, FCOUNT.

Frequency	Period	Scale Factor	Resolution
50 to 165 Hz.	9	221 to 67	3 Hz.
166 to 225 Hz.	5	121 to 89	3 Hz.
226 to 285 Hz.	3	147 to 117	3 Hz.
286 to 550 Hz.	2	175 to 91	6 Hz.

Figure 21. FREQ Output Ranges

```

/* COMPUTATION OF FREQUENCY RANGE */
57 3      IF FREQ < 285
          THEN DO;
59 4      IF FREQ < 226
          THEN DO;
61 5      IF FREQ < 166
          THEN RANGE = 9;
63 5      ELSE RANGE = 5;
64 5      END;
65 4      ELSE RANGE = 3;
66 4      END;
67 3      ELSE RANGE = 2;
/* LOAD MATH ACCUMULATOR WITH 100,000 */
68 3      CALL MQUID4 (.IR.,HUNDRED$K);
/* TEST FOR MOTOR SHUTDOWN */
69 3      IF FREQ > 1
          THEN DO;
71 4      /* DIVIDE BY FREQUENCY */
          CALL MQUIDV2 (.IR.,FREQ);
72 4      /* DIVIDE BY RANGE FACTOR */
          CALL MQUIDV1 (.IR.,RANGE);
73 4      /* GET TWO'S COMPLEMENT FOR iSBC 941 SCALE FACTOR */
          CALL MQUST1 (.IR.,FREQA);
74 4      FREQA=NOT (FREQA + 1);
75 4      END;

```

Figure 22. Period and Scale Factor Computations

The FCOUNT Primary Function is selected by sending the INITPF command followed by four parameters. The process is identical to that which was used in the previous example when we established the FREQ function. In this case, the sequence is described in the manual as is shown in Figure 23.

Description	Command/Data
Request INIT	C
Select FCOUNT	D
Input Select	D
Output Enable	D
Sampling Interval	D
Request PAUSE	C

Figure 23. FCOUNT Initialization

Let us examine the derivation of the terms which must make up the data table which will be transmitted to the processor in order to initialize it. The FCOUNT function does not allow the use of an external clock so we have no option as to which command will be sent to select this function. It is defined to be equal to 33H. This becomes the first element of the byte array used to contain the initial data.

The Input Select parameter describes which of the Port 1 inputs are to be measured. If we refer to Figure 13, we can see that a hardware assignment of Port 10 has been made for this function. This assignment corresponds to bit 0 of the parameter being set to a value of 1. The byte value for this parameter then becomes 01H.

The Output Enable byte is used to enable an output port corresponding with the input to change states when the Sampling Interval time has elapsed. Our system has a requirement to operate the control algorithm once each 200 milliseconds and we have previously indicated that the frequency counter would be used to establish this time interval. If the output is enabled and connected to an interrupt line, it will provide our system with the required pacer clock. The output bit from Port 20 will then be enabled to provide the interrupt. The parameter for this byte will be set to the same value as the Input Select and becomes 01H.

The Sampling Interval will establish the time interval to be used when sampling the input frequency. This time interval should be set to 200

milliseconds for our application. The parameter is then calculated from the equation:

$$\text{INTERVAL} = (\text{SAMPLE PERIOD}) / (0.02222) \\ \text{OR} \\ \text{INTERVAL} = (0.200) / (0.02222) = 9$$

The correct sampling interval for our control system should be set to a value of 09H.

A similar procedure can be used to send this data to the processor. The actual code used to implement the system can be found in Appendix A. Note that the unused bits of the device have been set to a predetermined value as was indicated by our hardware design of Figure 13.

Once the processor has been initiated and is performing its function, we need only wait until the device signals us that the 200 millisecond time interval has passed and that it is ready with the belt weight. When this interrupt occurs, we will read the data and perform our control functions. An interface must be established between the control algorithm and the processor which enables it to receive a value which represents the actual weight.

The total count received by the processor is available as a sixteen bit count made up of two eight bit bytes. The use of the Secondary Utility Commands, Read FCOUNT Measurements (RDFC0-RDFC1) allow the two bytes to be transferred into the host processor. We are using the first counter so we will use the corresponding commands, RDFC0 and RDFC1. An example of the procedure to read one of the count bytes can be seen in Figure 24.

The counter can be commanded to begin its next sample period by issuing a LOOP command to the processor. The two data bytes can be combined to form a 16-bit word and the resultant value divided by 2 to form a weight value. The division by two to obtain weight is required since the count range from 0 to 2000 corresponds to a weight of between 0 and 10.00 pounds; thus, each count has a value of 0.005 pounds. The integer numbers used in the control algorithm are fixed point with an implied scale factor of 100. The division by two provides a result which meets the criteria.

```

106 2 /* GET INPUT COUNT LOW BYTE */
107 3 DO WHILE ( (INPUT (UPI$STATUS) AND IBF) < > 0);
108 2 END;
      OUTPUT (UPI$COMMAND) = RDFC0;

109 2 DO WHILE ( (INPUT$STATUS) AND OBF) = 0);
110 3 END;
111 2 LCOUNT = INPUT (UPI$DATA);

```

Figure 24. FCOUNT Read Procedure

Appendix A provides the complete listing of the code which was used to interface with the processor assigned to the primary function, FCOUNT.

Stepper Motor Control Processor

The third example of utilizing the iSBC 941 Processor in an industrial application is provided by the processor installed into OBS socket 2. This device is used to drive a stepper motor which, in turn, controls the liquid valve position. Again, we will break the discussion into an initialization and an interface operational mode.

We find that the User's Guide indicates that initialization to the STEPPER Primary Function is performed by sending the INIT command followed by up to 21 data bytes. Figure 25 provides the table which shows the necessary parameters for this mode.

The technique used to place the processor into the desired function is the same as we have seen with the two other processors so we will not spend time dealing with the communications sequence. Instead, we will examine the techniques which can be used to determine the values of the initialization parameter bytes.

STEPPER is requested by sending a data byte of either 17H or 97H following the INIT command. Remember that the significance of setting bit 7 of the data high is to request that an external clock be used by the processor. There is no reason to use an external clock for our application, so we can choose a function request byte of 17H.

The remainder of the data is used to define the waveforms which are necessary to drive the stepper motor. We will derive the values for these parameters by beginning with the manufacturer's data sheet and moving until we have determined the correct value for each byte of data.

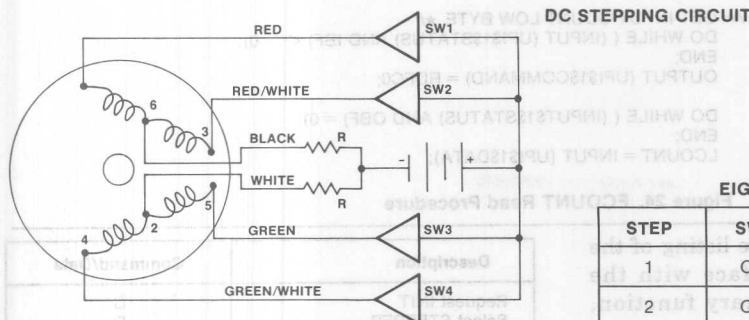
The motor chosen for this application utilizes four phases to drive the shaft. The data sheet provided

Description	Command/Data
Request INIT	C
Select STEPPER	D
Select Scale Factor	D
Output Enable	D
Output Polarity	D
Common Period	D
P20TRAN1	D
P20TRAN2	D
P21TRAN1	D
P21TRAN2	D
P22TRAN1	D
P22TRAN2	D
P23TRAN1	D
P23TRAN2	D
P24TRAN1	D
P24TRAN2	D
P25TRAN1	D
P25TRAN2	D
P26TRAN1	D
P26TRAN2	D
P27TRAN1	D
P27TRAN2	D
Request PAUSE	C

Figure 25. STEPPER Function Initialization

information for both a Four-Step Input Sequence (1.8 degrees per step) and for an Eight-Step Input Sequence (0.9 degrees per step). We will use the 1.8 degree step angles for our example and application. The data provided by the manufacturer is shown in Figure 26. The first task is to convert the switch state diagram into a desired waveform for each of the four phases. This has been done in Figure 27.

Beginning with Scale Factor, let us determine the required data parameters which will yield a stepper controller compatible with our motor. The Scale Factor will provide the minimum time period for one step to take place. The minimum time which we can specify is a function of both the motor characteristics and of the TRP for the primary function, STEPPER. The minimum TRP is determined by referencing the IDP User's Guide for the desired function. In this case, it is found to be $325 + (13 \times B)$ where B is the number of phases



FOUR-STEP INPUT SEQUENCE

STEP	SW1	SW2	SW3	SW4
1	ON	OFF	ON	OFF
2	ON	OFF	OFF	ON
3	OFF	ON	OFF	ON
4	OFF	ON	ON	OFF
5	ON	OFF	ON	OFF

EIGHT-STEP INPUT SEQUENCE

STEP	SW1	SW2	SW3	SW4
1	ON	OFF	ON	OFF
2	ON	OFF	OFF	OFF
3	ON	OFF	OFF	ON
4	OFF	OFF	OFF	ON
5	OFF	ON	OFF	ON
6	OFF	ON	OFF	OFF
7	OFF	ON	ON	OFF
8	OFF	OFF	ON	OFF
1	ON	OFF	ON	OFF

Figure 26. STEPPER Motor Input Sequence

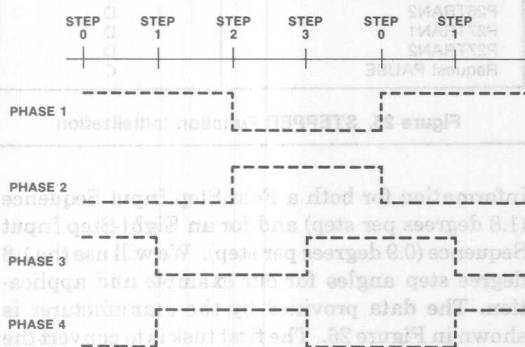


Figure 27. STEPPER Motor Waveforms

which are used. The result will be expressed in terms of processor cycles and can be converted into time by multiplying by 2.71 microseconds per cycle. This works out to be:

$$325 + (13 \times 4) = 377 \text{ PROCESSOR CYCLES}$$

OR

$$377 \times 2.71 = 1.021 \text{ MILLISECONDS}$$

Now, let's examine the minimum time which can be utilized by the stepper motor. This is given in the manufacturer's data sheets as being 2.86 milliseconds for the motor which we have chosen to

use. This value must be used to compute the Scale Factor for this application. The Scale Factor is computed by dividing the minimum step time by 86.72 microseconds or:

$$SF = 2.86 \text{ MILLISECONDS} / 86.72 \text{ MICROSECONDS} = 33$$

This number is entered into the processor using two's complement which becomes equal to 0DFH. The Output Enable is used to specify which of the eight possible control outputs are to be used to control the motor phases. The motor phase assignments to I/O ports was made in Figure 15 and indicates that Ports 24 through 27 will be enabled for the primary function. Setting the corresponding bits provides a parameter to be sent to the processor of 0F0H.

The rest of the parameters deal with providing a definition of the waveforms generated in Figure 26 to the processor. The following paragraphs deal with the operations required to convert the graphic representation into data parameters.

Each phase must be initialized to an initial output state which corresponds to the signal level shown for Step 0 of Figure 27. A "1" will be placed into the bit corresponding to each of the port's output bits which are to be in a logical one state upon

reaching step 0. We see that Bits 24 and 26 are set corresponding to phase 1 and 3. The data byte for Initial Output is thus defined to be 050H.

The Period parameter for a stepper motor function corresponds to the number of steps which are defined in the motor's step sequence. Our example uses a four step sequence so the Common Period will be set to a value of 04H.

The remainder of the initialization parameters define the transitions of each of the phases. This involves the examination of the waveform and noting the points at which the output level changes. This data can be input to allow the device to accurately produce the control waveforms for any stepper motor control mode. We are not using the first four output bits so the transition definitions for these outputs is meaningless and will be output as zeroes. The waveform for output Port 24 shows a transition at steps 1 and 3. The parameter for the first transition of Port 24, P24TRAN1 is defined to be 00H. Likewise, the second transition, P24TRAN2 is set to a value of 02H.

The technique used above can be continued to define the constants, P25TRAN1 and P25TRAN2 as being the same as for Port 24 or 00H and 02H respectively.

The transitions for the phases driven from Port 26 and 27 can be seen to occur at steps 1 and 3 so the data for those parameters can easily be seen to be set to 01H and 03H for each port.

The initialization table can be sent to the processor using the same techniques as were used

for the processors discussed previously. The complete program for the initialization can be found in Appendix A.

A driver must next be prepared which will be used to provide the interface between the control algorithm and the IDP processor which supports the stepper motor. When the STEPPER primary function is used, a queue is utilized for supporting the step commands to the motor. Each command to the stepper consists of a data byte signifying the step rate to be used and a data byte which provides the signed magnitude of the number of steps to be moved. Using the motor to control a flow control valve allows us to use a constant step rate, but some type of program must be prepared which will convert the signed two's complement representation of the position from the control algorithm to a signed magnitude format.

The number conversion is easily done and the PL/M-80 programming code to perform the format change is shown in Figure 28.

The data queue allows up to six movement commands to be present and waiting to be serviced by the IDP. If the processor is behind in its operations and cannot accept a seventh request, the host must wait until one of the requests in the queue has been serviced. The queue status bits can be tested to determine if room exists for another command and the "queue not empty" bit can be tested to verify that all requested movements have been completed. Normal operation of our motor should be such that the queue is not allowed to fill to its maximum capacity.

```

141  3  /* SUPPORT CONVERSION TO SIGNED MAGNITUDE NUMBER */
      IF POSITION > 127
      THEN DO;

143  4  /* GET MAGNITUDE OF MOVEMENT */
      POSITION = 256 - POSITION;

144  4  /* SET SIGN FOR CCW ROTATION */
      POSITION = POSITION OR REVERSE;
145  4  END;

```

Figure 28. Number Format Conversion

The code which is required to test the queue and to send a stepper movement request is shown in Figure 29. The complete code can be seen in Appendix A.

VIII. APPLICATION SOFTWARE

Having developed the software which is required to support the Industrial Digital Processors, we can now devote our time to the task of implementing the application software and of handling any programs which are required to support functions unique to the host iSBC 569 board. This software can be grouped into two general categories, initialization programs, and control algorithm programs.

Initialization Programs

The initialization of the iSBC 569 involves setting up the required configuration of interrupt handling and of the devices which are installed into the slave sockets. For the purposes of this application, we will include some system diagnostic capabilities within the process. These routines will be executed each time a RESET or a POWER-UP occurs. Only the highlights of the code used will be presented in detail; however, the complete listings of the initialization programs can be found in Appendix A by referring to the BCKGND Program listing.

A unique feature of using the iSBC 941 processors is their ability to provide, upon request, an

identification code. The initiation diagnostic program takes advantage of this fact by interrogating each processor and verifying that the correct ID code is returned. If any of the processors have failed catastrophically or if the internal data bus of the host board has failed, the program will provide an indication of this fact.

Each of the slave processors has, associated with it, an individual hardware reset line which is under the control of the host. A reset or power up condition will cause the control lines to reset to the state which hold each slave in a reset state. Before any slave can be used, it's associated reset line must be de-activated. This is done by sending a logical one to the corresponding bit of the Reset Latch. Other bits of the Reset Latch can be used to illuminate the on-board LED or to generate an interrupt to another board on the Multibus data bus.

A special PL/M-80 command is utilized to disable the reset interrupts of the 8085A host processor. Execution of this command will allow all serviceable interrupts to enter via the 8259A Interrupt Controller. The command which will mask off the unused interrupt structure is shown in Figure 30. The initialization process must also initialize the FSP Integer Record. This will allow the use of the math support routines which will be required to support the control algorithm.

```

146 3  /* VERIFY THAT QUEUE SPACE IS AVAILABLE */
147 4  DO WHILE ( (INPUT (UPI$2$STATUS) AND QF) < > 0);
      END;

      /* REQUEST DESIRED STEP RATE */
148 3  DO WHILE ( (INPUT (UPI$2$STATUS) AND IBF) < > 0);
149 4  END;
150 3  OUTPUT (UPI$2$DATA) = STEP$RATE;

      /* REQUEST STEPPER MOVEMENT */
151 3  DO WHILE ( (INPUT (UPI$2$STATUS) AND IBF) < > 0);
152 4  END;
153 3  OUTPUT (UPI$DATA) = POSITION;

```

Figure 29. STEPPER Movement Request

```

34 1  /* MASK OUT THE RESET INTERRUPTS OF THE PROCESSOR */
      CALL S$MASK (MASKS);

```

Figure 30. PL/M-80 Sim Instruction

Control Algorithm Programs

The program which actually handles the control algorithm for the two loops can be found in Appendix A, MAIN\$CONTROL. The flow of the program is straightforward and can easily be followed by reading the listing. The operations are primarily handled by the use of repeated calls to the FSP integer math routines and to the processor interface modules which we have previously generated.

It is beyond the scope of this application note to dwell upon the techniques which were used to generate the PID control routine; this aspect will be covered in a future application note.

Limits were placed upon the control outputs so that the signals to the processors would not exceed the physical limits of the external devices. For example, the frequency range is limited to range between 0 and 550 to correspond with the operating range of the weighbelt as we have defined it. The limits upon the liquid control valve have been set at plus and minus 10 steps since this is the maximum distance which the stepper motor can travel in any one 200 millisecond time period; increasing the possible count could result in filling the queue. This could cause the 200 millisecond time to be extended if we had to wait for the queue to empty.

Master Processor

A complete control solution to the weighbelt feeder and the liquid applicator has now been developed. The process is stand alone and resides entirely upon a single board. It can operate without requiring any access from the MULTIBUS bus, thus freeing the bus for other control, monitoring or supervisory duties.

The system developed for this application note requires a setpoint for the mass flow and a liquid ratio be provided to the control system. This information would normally be supplied by some type of bus master device. We have chosen to use the pre-configured RMX/80 BASIC-80 Interpreter to perform this task. A simple program needs to be prepared which will allow adjustment of the setpoints and monitoring of the operation of the control system.

Using BASIC will provide full disk I/O capabilities to the operator. Communicating with the

system through a CRT terminal, he can write and execute programs which use the resources of the system disk or of any of the controllers which may be present on the bus.

Two programs are required to perform this task. Since they are written in BASIC, they may easily be modified or expanded if the need should ever arise. Indeed, other programs could be written to perform other tasks, such as optimizing the control parameters.

In both programs, the parameters involved with the control operation are accessed by using the PEEK and POKE instructions. Remember that the iSBC 569 controller allows the on-board memory to be made available to other devices on the bus through the dual port mechanism. In our application, this has been done by jumpering the board such that the on-board memory beginning at location 8000H can be accessed on the bus at location 2000H. This mapping was done since the memory locations at 2000H are not used by BASIC unless requested to do so. A byte of data which is at location 827EH on the controller can be read by performing a PEEK of location 227EH. Some of the memory assignments for the controller have been shown in Figure 31.

MOD MAINCONTROLMODULE		
829 FH	SYM	MEMORY
823 3H	SYM	PRLO
825 FH	SYM	CONSTANT\$1
00DCH	SYM	BOUNDS2
00E 6H	SYM	TIMEINTERVAL
827 AH	SYM	LIQUIDFLOW
00E 8H	SYM	DISTREV
828 0H	SYM	MASSFLOW
828 5H	SYM	LIQUIDVALVE
828 8H	SYM	DUMMY
00E FH	SYM	ZERO
01ADH	SYM	PIDRUN
81F 7H	SYM	IR
825DH	SYM	LIQCOUNT
826 8H	SYM	CONSTANTS2
00E 4H	SYM	CONTROL1
827 7H	SYM	BELTSPEED
827 CH	SYM	MASSSETPOINT
00E 9H	SYM	CONVLENGTH
828 2H	SYM	BELTCONTROL
828 7H	SYM	SYSTEMRUNNING
828 AH	SYM	ICW
3F0 0H	SYM	JUMPTABLE
820 9H	SYM	PRCV
825 EH	SYM	BELTCOUNT
00D 4H	SYM	BOUNDS1
00E 5H	SYM	CONTROL2
827 8H	SYM	BELTWEIGHT
827 EH	SYM	SETPOINT
00E AH	SYM	SIX
828 4H	SYM	LIQUIDRATIO
00E BH	SYM	ERRORFIELD
00EDH	SYM	THOUSAND
00F 1H	SYM	INITIATION

Figure 31. Selected Memory Location Assignments

The first program involves setting up the two control parameters and handling the control flag which causes the process to start and to stop. This program can be found in Figure 32.

```

10 REM THIS PROGRAM IS USED TO INPUT SETPOINTS
15 REM TO THE LIQUID CONTROL SYSTEM.
20 POKE 02287H,0
25 INPUT "ENTER MASS SETPOINT-";MS
26 IF MS > 1200 THEN 25
30 MS=CINT(MS*10/60)
35 H=INT(MS/256)
40 L=CINT(MS-H*256)
45 POKE 0227EH,L
50 POKE 0227FH,H
55 INPUT "PERCENT LIQUID-";LR
60 LR=CINT(LR)
65 IF LR > 127 THEN 55
70 POKE 02284H,LR
75 POKE 02287H,1
80 RUN "STATUS"

```

Figure 32. Basic Program for Parameter Initialization

PROGRAM NAME: STATUS

```

10 H=PEEK(0227EH)
20 H=PEEK(0227FH)
30 MS=((256*H)+L)/60/10
40 L=PEEK(02278H)
50 H=PEEK(02279H)
60 WT=((256*H)+L)/100
70 L=PEEK(022890H)
80 H=PEEK(02281H)
90 AM=((256*H)+L)/60/10
100 MT=PEEK(02294H)
110 LR=(PEEK(02284H))/100
120 LS=AM*LR
130 L=PEEK(0227AH)
140 H=PEEK(0227BH)
150 LF=((256*H)+L)/100
160 PRINT "MASS SETPOINT","WEIGHT","ACTUAL MASS","MOTION"
170 PRINT MS,WT,AM,MT
180 PRINT "LIQUID RATIO","LIQUID SET","LIQUID FLOW"
190 PRINT LR,LS,LF
200 Z=PEEK(02285H)
210 IF Z < 128 THEN 230
220 Z=256-Z
225 Z=0-Z
230 L=PEEK(02282H)
231 H=PEEK(02283H)
232 BS=((256*H)+L)/60/200
239 PRINT "STEPPER";Z, "BELT";BS
240 PRINT " "
250 PRINT " "
260 FOR N=0 TO 1000
270 NEXT N
280 GO TO 10

```

Figure 33. Basic Snapshot Program

Upon completion of the initialization program, a second program provides a display of the system operation. This program could have been an optional program which is only called when the operator desires to view the system operation. A program which provides a snapshot of the system operation is shown in Figure 33. Again, the program is interactive with the operator and can easily be modified at any time to reformat or display additional information.

IX. CONCLUSION

The purpose of this application note has been to demonstrate some of the techniques which can be used to provide a control system design solution using an intelligent slave concept. This has been done and the system has been constructed and has been found to operate as the design specified. The Intelligent Slave Concept does provide a single board solution to distributed control and certainly off-loads the master processor of control duties.

This frees the master to provide supervisory control and human interface duties.

Certainly, this concept can be expanded to encompass a broad variety of complex control

situations. At the same time, there is no reason why the Intelligent Slave board could not be used to provide a single board solution to a simple control application where no interaction with other processes is required.

APPENDIX A

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE BACKGROUNDMODULE
 OBJECT MODULE PLACED IN :F1:BCKGND.OBJ
 COMPILER INVOKED BY: PLM80 :F1:BCKGND.PLM DEBUG PAGEWIDTH(72) TITLE('BA
 -CKGROUND PROGRAM')

```

/*****
* THIS IS THE MAIN BACKGROUND OPERATING
* PROGRAM FOR THE PID CONTROL SYSTEM.
*****/

1      BACKGROUND$MODULE: DO;

/* DECLARATION OF BOARD I/O ASSIGNMENTS */
2      1      DECLARE UPI$0$STATUS      LITERALLY '0E5H';
3      1      DECLARE UPI$1$STATUS      LITERALLY '0E7H';
4      1      DECLARE UPI$2$STATUS      LITERALLY '0E9H';

5      1      DECLARE UPI$0$COMMAND     LITERALLY '0E5H';
6      1      DECLARE UPI$1$COMMAND     LITERALLY '0E7H';
7      1      DECLARE UPI$2$COMMAND     LITERALLY '0E9H';

8      1      DECLARE UPI$0$DATA        LITERALLY '0E4H';
9      1      DECLARE UPI$1$DATA        LITERALLY '0E6H';
10     1      DECLARE UPI$2$DATA        LITERALLY '0E8H';

11     1      DECLARE RESET$LATCH$ADR    LITERALLY '0EAH';

/* DECLARATION OF RAM TEST PARAMETERS */
12     1      DECLARE BEGIN$RAM          LITERALLY '8000H';
13     1      DECLARE END$RAM            LITERALLY '8500H';
14     1      DECLARE ZERO$PATTERN       LITERALLY '000H';
15     1      DECLARE ONE$PATTERN        LITERALLY '0FFH';

/* DECLARATION OF RESET LATCH BIT ASSIGNMENTS */
16     1      DECLARE RESET$UPI$0        LITERALLY '00000001B';
17     1      DECLARE RESET$UPI$1        LITERALLY '00000010B';
18     1      DECLARE RESET$UPI$2        LITERALLY '00000100B';
19     1      DECLARE LIGHT$LED          LITERALLY '00001000B';
20     1      DECLARE MULTI$INTR         LITERALLY '00010000B';

/* DECLARATION OF ISBC 941 STATUS BITS */
21     1      DECLARE IBF                LITERALLY '00000010B';
22     1      DECLARE OBF                LITERALLY '00000001B';

/* DECLARATION OF ISBC 941 COMMANDS */
23     1      DECLARE IDEN               LITERALLY '000H';

/* DECLARATION OF ISBC 941 IDENTIFICATION CODE */
24     1      DECLARE SBC941             LITERALLY '41H';

/* DECLARATION OF MEMORY TEST ADDRESS REGISTER */
25     1      DECLARE I ADDRESS AT (87FEH);
26     1      DECLARE MEMLOC BASED I BYTE;

/* DECLARATION OF RESET MASKS FOR 8085 PROCESSOR */

```

```

27 1 DECLARE MASKS BYTE DATA (00FH);
/* DECLARATION OF PL/M-80 SIM INSTRUCTION */
28 1 S$MASK: PROCEDURE (MASK) EXTERNAL;
29 2 DECLARE MASK BYTE;
30 2 END S$MASK;
/* DECLARATION OF INITIATION TASK */
31 1 INITIATION:
/* PROCEDURE EXTERNAL;
32 2 END INITIATION;
/* CLEAR ISBC 941 DEVICES USING ON-BOARD RESET */
33 1 OUTPUT (RESET$LATCH$ADR) = 0;
/* MASK OUT THE RESET INTERRUPTS OF THE PROCESSOR */
34 1 CALL S$MASK (MASKS);
/* TEST MEMORY RAM LOCATIONS */
35 1 DO I = BEGIN$RAM TO END$RAM;
36 2 MEMLOC = ZERO$PATTERN;
37 2 DO WHILE MEMLOC <> ZERO$PATTERN;
38 3 END;
39 2 MEMLOC = ONE$PATTERN;
40 2 DO WHILE MEMLOC <> ONE$PATTERN;
41 3 END;
42 2 END;
/* RELEASE 941 LOCKOUT/RESET BITS */
43 1 OUTPUT (RESET$LATCH$ADR) = RESET$UPI$0 OR
RESET$UPI$1 OR
RESET$UPI$2 OR
MULTI$INTR;
/* VERIFY THAT SBC941 PROCESSOR IS IN SOCKET 0 */
44 1 DO WHILE ((INPUT (UPI$0$STATUS) AND IBF) <> 0);
45 2 END;
46 1 OUTPUT (UPI$0$COMMAND) = IDEN;
47 1 DO WHILE ((INPUT (UPI$0$STATUS) AND OBF) = 0);
48 2 END;
49 1 DO WHILE (INPUT (UPI$0$DATA) <> SBC941);
50 2 END;
/* VERIFY THAT SBC941 PROCESSOR IS IN SOCKET 1 */
51 1 DO WHILE ((INPUT (UPI$1$STATUS) AND IBF) <> 0);
52 2 END;
53 1 OUTPUT (UPI$1$COMMAND) = IDEN;
54 1 DO WHILE ((INPUT (UPI$1$STATUS) AND OBF) = 0);
55 2 END;
56 1 DO WHILE (INPUT (UPI$1$DATA) <> SBC941);
57 2 END;

```

```

58      1      /* VERIFY THAT SBC941 PROCESSOR IS IN SOCKET 2 */
59      2      DO WHILE ((INPUT (UPI$2$STATUS) AND IBF) <> 0);
60      1      END;
61      1      OUTPUT (UPI$2$COMMAND) = IDEN;
62      2      DO WHILE ((INPUT (UPI$2$STATUS) AND OBF) = 0);
63      1      END;
64      2      DO WHILE (INPUT (UPI$2$DATA) <> SBC941);
        1      END;

        /* START-UP TEST OK- TURN OFF LED */
65      1      OUTPUT (RESET$LATCH$ADR) = RESET$UPI$0 OR
        RESET$UPI$1 OR
        RESET$UPI$2 OR
        LIGHT$LED OR
        MULTI$INTR;

        /* INITIATE THE CONTROL DEVICES */
66      1      CALL INITIATION;

        /* PERFORM BACKGROUND TASKS */
67      1      DO WHILE 1;
68      2      HALT;
69      2      END;

70      1      END BACKGROUND$MODULE;

MODULE INFORMATION:
CODE AREA SIZE      = 00D4H 212D
VARIABLE AREA SIZE  = 0000H   0D
MAXIMUM STACK SIZE  = 0002H   2D
128 LINES READ
0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION

```

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE MAINCONTROLMODULE
 OBJECT MODULE PLACED IN :F1:CNITTSK.OBJ
 COMPILER INVOKED BY: PLM80 :F1:CNITTSK.PLM DEBUG

```

$INTVECTOR(4,3F00H)
$PAGEWIDTH(72)
$TITLE('MAIN CONTROL')
/*****
*
*      MAIN$CONTROL$TASK
* THIS TASK IS USED TO CONTROL THE TWO PID CONTROL
* LOOPS. ONE LOOP CONTROLS THE SPEED OF A CONVEYOR
* WHILE THE SECOND CONTROLS THE FLOW OF A LIQUID.
* THE TASK OPERATES EACH 200 MSEC.
*
*      ***** VERSION 1.1 *****
1      MAIN$CONTROL$MODULE: DO;
      /* DECLARATION OF PID RECORD SET-UP TASK */
2      1      UQPSET:
          PROCEDURE (PR$PTR,ERROR$FLD$PTR,PRIV$PTR) EXTERNAL
          ;
3      2      DECLARE (PR$PTR,ERROR$FLD$PTR,PRIV$PTR) ADDRESS;
4      2      END UQPSET;

      /* DECLARATION OF PID CONTROL BITS */
5      1      UQPSCT:
          PROCEDURE (PR$PTR,CONTROL$PTR) EXTERNAL;
6      2      DECLARE (PR$PTR,CONTROL$PTR) ADDRESS;
7      2      END UQPSCT;

      /* PROCEDURE TO SET UP PID CONSTANTS */
8      1      UQPSCN:
          PROCEDURE (PR$PTR,CONSTANT$PTR) EXTERNAL;
9      2      DECLARE (PR$PTR,CONSTANT$PTR) ADDRESS;
10     2      END UQPSCN;

      /* DEFINE THE DEFAULT ERROR HANDLER */
11     1      UQPSBD:
          PROCEDURE (PR$PTR,BOUND$PTR) EXTERNAL;
12     2      DECLARE (PR$PTR,BOUND$PTR) ADDRESS;
13     2      END UQPSBD;

      /* PROCEDURE TO CHANGE THE TIME INTERVAL */
14     1      UQPSTI:
          PROCEDURE (PR$PTR,TIME$INTERVAL$PTR) EXTERNAL;
15     2      DECLARE (PR$PTR,TIME$INTERVAL$PTR) ADDRESS;
16     2      END UQPSTI;

      /* DECLARATION OF THE PID CONTROL PROGRAM */
17     1      UQPPID:
          PROCEDURE (PR$PTR,IR$PTR) EXTERNAL;
18     2      DECLARE (PR$PTR,IR$PTR) ADDRESS;
19     2      END UQPPID;

```

```

20 1      /* DECLARATION OF WEIGHBELT SPEED INTERFACE */
      WEIGHBELT$SPEED:
21 2      PROCEDURE BYTE EXTERNAL;
      END WEIGHBELT$SPEED;

      /* DECLARATION OF WEIGHBELT WEIGHT INTERFACE */
22 * 1      WEIGHBELT$WEIGHT:
      PROCEDURE ADDRESS EXTERNAL;
23 * 2      END WEIGHBELT$WEIGHT;

      /* DECLARATION OF LIQUID FLOW RATE INTERFACE */
24 * 1      LIQUID$FLOW$RATE:
      PROCEDURE ADDRESS EXTERNAL;
25 2      END LIQUID$FLOW$RATE;

      /* DECLARATION OF WEIGHBELT MOTOR DRIVE INTERFACE */
26 1      WEIGHBELT$MOTOR$DRIVE:
      PROCEDURE (SPEED) EXTERNAL;
27 2      DECLARE SPEED ADDRESS;
28 2      END WEIGHBELT$MOTOR$DRIVE;

      /* DECLARATION OF LIQUID VALVE INTERFACE */
29 1      LIQUID$VALVE$POSITION:
      PROCEDURE (POSITION) EXTERNAL;
30 2      DECLARE POSITION BYTE;
31 2      END LIQUID$VALVE$POSITION;

      /* DECLARATION OF PROCESSOR 0 INITIALIZATION MODULE */
32 1      PROCESSOR$0$INITIALIZATION:
      PROCEDURE EXTERNAL;
33 2      END PROCESSOR$0$INITIALIZATION;

      /* DECLARATION OF PROCESSOR 1 INITIALIZATION MODULE */
34 1      PROCESSOR$1$INITIALIZATION:
      PROCEDURE EXTERNAL;
35 2      END PROCESSOR$1$INITIALIZATION;

      /* DECLARATION OF PROCESSOR 2 INITIALIZATION MODULE */
36 1      PROCESSOR$2$INITIALIZATION:
      PROCEDURE EXTERNAL;
37 2      END PROCESSOR$2$INITIALIZATION;

      /* DECLARATION OF PIT COUNTER 1 INITIALIZATION */
38 1      COUNTER$1$INITIALIZATION:
      PROCEDURE EXTERNAL;
39 2      END COUNTER$1$INITIALIZATION;

      /* DECLARATION OF PIT COUNTER 2 INITIALIZATION */
40 1      COUNTER$2$INITIALIZATION:
      PROCEDURE EXTERNAL;
41 2      END COUNTER$2$INITIALIZATION;

```



```

/* DECLARATION OF FSP UNSIGNED LOAD PROCEDURES */
42 1 MQULD1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
43 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
44 2 END MQULD1;
45 1 MQULD2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
46 2 *****DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
47 2 * END MQULD2;
/* DECLARATION OF FSP UNSIGNED MULTIPLY PROCEDURE */
48 1 MQUML1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
49 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
50 2 END MQUML1;
51 1 MQUML2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
52 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
53 2 END MQUML2;
/* DECLARATION OF FSP UNSIGNED DIVIDE PROCEDURE */
54 1 MQUDV1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
55 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
56 2 END MQUDV1;
57 1 MQUDV2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
58 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
59 2 END MQUDV2;
/* DECLARATION OF FSP SIGNED DIVIDE PROCEDURE */
60 1 MQSDV1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
61 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
62 2 END MQSDV1;
63 1 MQSDV2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
64 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
65 2 END MQSDV2;
/* DECLARATION OF FSP SIGNED STORE PROCEDURE */
66 1 MQSST2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
67 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
68 2 END MQSST2;
/* DECLARATION OF FSP SIGNED LOAD PROCEDURE */
69 1 MQSLD2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
70 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
71 2 END MQSLD2;
/* DECLARATION OF FSP SIGNED SUBTRACT PROCEDURE */
72 1 MQSSB2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
73 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
74 2 END MQSSB2;
/* DECLARATION OF FSP UNSIGNED STORE PROCEDURE */
75 1 MQUST1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
76 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
77 2 END MQUST1;
78 1 MQUST2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
79 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
80 2 END MQUST2;

```

```

/* DECLARATION OF FSP SIGNED MULTIPLY PROCEDURE */
81 1 MQSML1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
82 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
83 2 END MQSML1;
$EJECT
/*****
* DATA STORAGE AREAS FOR THE PID CONTROL *
*****/

/* DEFINITION OF LIMITATION CONSTANTS */
84 1 DECLARE MAX$MOTOR$SPEED LITERALLY '550';
85 1 DECLARE MIN$MOTOR$SPEED LITERALLY '0';
86 1 DECLARE MAX$VALVE$MOVEMENT LITERALLY '10';
87 1 DECLARE MIN$VALVE$MOVEMENT LITERALLY '-10';

/* DEFINITION OF PID PARAMETER COEFFICIENTS */
88 1 DECLARE FEEDER$C0 LITERALLY '1';
89 1 DECLARE FEEDER$C1 LITERALLY '1';
90 1 DECLARE FEEDER$C2 LITERALLY '1';
91 1 DECLARE FEEDER$C3 LITERALLY '1';
92 1 DECLARE FEEDER$TIME$INTERVAL LITERALLY '1';
93 1 DECLARE FEEDER$SCALE$FACTOR LITERALLY '1';

94 1 DECLARE LIQUID$C0 LITERALLY '1';
95 1 DECLARE LIQUID$C1 LITERALLY '1';
96 1 DECLARE LIQUID$C2 LITERALLY '1';
97 1 DECLARE LIQUID$C3 LITERALLY '1';
98 1 DECLARE LIQUID$TIME$INTERVAL LITERALLY '1';
99 1 DECLARE LIQUID$SCALE$FACTOR LITERALLY '10';

/* DEFINITION OF RESET LATCH PARAMETERS */
100 1 DECLARE RESET$LATCH$ADR LITERALLY '0EAH';
101 1 DECLARE INDICATOR$ON LITERALLY '07H';
102 1 DECLARE INDICATOR$OFF LITERALLY '0FH';

/* RESERVE 18 BYTES FOR THE INTEGER RECORD */
103 1 DECLARE IR (18) BYTE PUBLIC;

/* RESERVE 42 BYTES FOR EACH PID RECORD */
104 1 DECLARE PRCV (42) BYTE;
105 1 DECLARE PRLQ (42) BYTE;

/* RESERVE SPACE FOR COUNTER DATA */
106 1 DECLARE (LIQ$COUNT,BELT$COUNT) BYTE PUBLIC;

/* RESERVE 12 BYTES FOR EACH CONSTANT ARRAY */
107 1 DECLARE CONSTANTS1 STRUCTURE (
C0 ADDRESS,
C1 ADDRESS,
C2 ADDRESS,
C3 ADDRESS,
DT ADDRESS,
S ADDRESS);

```

```

108 1      DECLARE CONSTANTS2 STRUCTURE (
          C0 ADDRESS,
          C1 ADDRESS,
          C2 ADDRESS,
          C3 ADDRESS,
          DT ADDRESS,
          S ADDRESS );

          /* RESERVE 8 BYTES FOR EACH BOUNDS ARRAY */
109 1      DECLARE BOUNDS1 (4) ADDRESS DATA (
          000H,
          000H,
          MAX$MOTOR$SPEED,
          MIN$MOTOR$SPEED );

110 1      DECLARE BOUNDS2 (4) ADDRESS DATA (
          000D,
          000D,
          MAX$VALVE$MOVEMENT,
          MIN$VALVE$MOVEMENT );

          /* RESERVE 1 BYTE FOR EACH CONTROL BYTE */
111 1      DECLARE CONTROL1 BYTE DATA (073H);
112 1      DECLARE CONTROL2 BYTE DATA (053H);

          /* DECLARE TIME INTERVAL */
113 1      DECLARE TIME$INTERVAL ADDRESS DATA (1);

          /* RESERVE SPACE FOR THE CURRENT BELT SPEED */
114 1      DECLARE BELT$SPEED BYTE;

          /* RESERVE SPACE FOR THE CURRENT BELT WEIGHT */
115 1      DECLARE BELT$WEIGHT ADDRESS;

          /* RESERVE SPACE FOR THE LIQUID FLOW */
116 1      DECLARE LIQUID$FLOW ADDRESS;

          /* RESERVE SPACE FOR THE EFFECTIVE SETPOINT */
117 1      DECLARE MASS$SETPOINT ADDRESS;

          /* RESERVE SPACE FOR THE DESIRED SETPOINT */
118 1      DECLARE SET$POINT ADDRESS;

          /* RESERVE SPACE FOR THE DISTANCE OF BELT PER REVOLUTION
          - */
119 1      DECLARE DIST$REV BYTE DATA (100);

          /* DEFINE THE CONVEYOR LENGTH */
120 1      DECLARE CONV$LENGTH BYTE DATA (200);

          /* DEFINE THE CONSTANT SIX */
121 1      DECLARE SIX BYTE DATA (6);

          /* RESERVE STORAGE FOR ACTUAL CURRENT MASS FLOW */
122 1      DECLARE MASS$FLOW ADDRESS;

```

```

123 1      /* RESERVE SPACE FOR BELT CONTROL OUTPUT */
          DECLARE BELT$CONTROL ADDRESS;

124 1      /* RESERVE SPACE FOR LIQUID RATIO */
          DECLARE LIQUID$RATIO BYTE;

125 1      /* RESERVE SPACE FOR LIQUID CONTROL OUTPUT */
          DECLARE LIQUID$VALVE ADDRESS;

126 1      /* RESERVE SPACE FOR RUN/HAUT CONTROL */
          DECLARE SYSTEM$RUNNING BYTE PUBLIC;

127 1      /* RESERVE SPACE FOR ERROR FIELD */
          DECLARE ERROR$FIELD ADDRESS DATA (0F800H);
128 1      DECLARE DUMMY ADDRESS;

129 1      /* RESERVE SPACE FOR PIC ICW BYTE */
          DECLARE ICW BYTE;

130 1      /* DEFINE CONSTANT 1000 */
          DECLARE THOUSAND ADDRESS DATA (1000);

131 1      /* DEFINE CONSTANT 0 */
          DECLARE ZERO ADDRESS DATA (0);

132 1      /* DEFINE INTERRUPT JUMP TABLE */
          DECLARE JUMP$TABLE BYTE AT (3F00H);

133 1      /* DECLARATION OF PIC ADDRESSES ON ISBC 569 BOARD */
          DECLARE PIC$ICW1$PTR LITERALLY '0ECH';
134 1      DECLARE PIC$ICW2$PTR LITERALLY '0EDH';
135 1      DECLARE PIC$INT$MASK$PTR LITERALLY '0EDH';

          /* DECLARATION OF PIC CONSTANTS */
136 1      DECLARE CLR$LOW$BITS LITERALLY '0E0H';
137 1      DECLARE INTERVAL$4 LITERALLY '016H';
138 1      DECLARE INTERRUPT$MASK LITERALLY '0F4H';

          $EJECT
          /*****
          * INITIALIZE PROGRAM AT START-UP OF SYSTEM *
          * THIS PROCEDURE IS CALLED AT START-UP *
          *****/

139 1      INITIATION: PROCEDURE PUBLIC;

140 2      /* DISABLE THE INTERRUPTS */
          DISABLE;

          /* INITIALIZE PID RECORD */
141 2      CALL UQPSET (.PRCV,.ERROR$FIELD,.DUMMY);
142 2      CALL UQPSET (.PRLQ,.ERROR$FIELD,.DUMMY);

```

		/* INITIALIZE THE CONTROL BITS */	3	173
143	2	CALL UQPSCT (.PRCV,.CONTROL1);	3	174
144	2	CALL UQPSCT (.PRLQ,.CONTROL2);		
		/* SET UP THE PID CONSTANTS */	3	175
145	2	CONSTANTS1.C0 = FEEDER\$C0;		
146	2	CONSTANTS1.C1 = FEEDER\$C1;		
147	2	CONSTANTS1.C2 = FEEDER\$C2;	3	176
148	2	CONSTANTS1.C3 = FEEDER\$C3;		
149	2	CONSTANTS1.DT = FEEDER\$TIME\$INTERVAL;		
150	2	CONSTANTS1.S = FEEDER\$SCALE\$FACTOR;	3	177
			3	178
151	2	CONSTANTS2.C0 = LIQUID\$C0;		
152	2	CONSTANTS2.C1 = LIQUID\$C1;		
153	2	CONSTANTS2.C2 = LIQUID\$C2;		
154	2	CONSTANTS2.C3 = LIQUID\$C3;		
155	2	CONSTANTS2.DT = LIQUID\$TIME\$INTERVAL;		
156	2	CONSTANTS2.S = LIQUID\$SCALE\$FACTOR;		
		/* CLEAR SETPOINTS */		
157	2	SETPOINT = 0;		
158	2	LIQUID\$RATIO = 0;		
159	2	SYSTEM\$RUNNING = 0;	1	179
		/* INITIALIZE THE CONSTANTS */		
160	2	CALL UQPSCN (.PRCV,.CONSTANTS1);	3	180
161	2	CALL UQPSCN (.PRLQ,.CONSTANTS2);		
		/* INITIALIZE THE BOUNDS */	3	181
162	2	CALL UQPSBD (.PRCV,.BOUNDS1);		
163	2	CALL UQPSBD (.PRLQ,.BOUNDS2);	3	182
		/* SET THE TIME INTERVAL */		
164	2	CALL UQPSTI (.PRCV,.TIME\$INTERVAL);		
165	2	CALL UQPSTI (.PRLQ,.TIME\$INTERVAL);	3	183
		/* INITIALIZE PROCESSOR 0 */	3	184
166	2	CALL PROCESSOR\$0\$INITIALIZATION;		
		/* INITIALIZE PROCESSOR 1 */	3	185
167	2	CALL PROCESSOR\$1\$INITIALIZATION;	3	186
		/* INITIALIZE PROCESSOR 2 */	3	187
168	2	CALL PROCESSOR\$2\$INITIALIZATION;	3	188
		/* INITIALIZE COUNTER 1 */	3	189
169	2	CALL COUNTER\$1\$INITIALIZATION;		
		/* INITIALIZE COUNTER 2 */	3	190
170	2	CALL COUNTER\$2\$INITIALIZATION;	3	191
		/* INITIALIZE INTERRUPT CONTROLLER */		
171	2	ICW = (LOW (.JUMP\$TABLE) AND CLR\$LOW\$BITS) OR INTERVAL\$4;	3	192
172	2	OUTPUT (PIC\$ICW1\$PTR) = ICW;	3	193


```

173 2      ICW = HIGHC(.JUMP$TABLE);
174 2      OUTPUT (PIC$ICW2$PTR) = ICW;
      /* SET INTERRUPT MASKS */
175 2      OUTPUT (PIC$INT$MASK$PTR) = INTERRUPT$MASK;
      /* ENABLE INTERRUPTS */
176 2      ENABLE;
      /* RETURN TO MAIN PROGRAM */
177 2      RETURN;
178 2      END INITIATION;
      $EJECT
      /*****
      * THIS IS THE PID CONTROL ROUTINE. IT IS ENTERED *
      * EACH 200 MILLISECONDS THROUGH AN INTERRUPT GEN- *
      * ERATED BY THE FREQUENCY COUNTER UPI AND SENT TO *
      * INTERRUPT 3. *
      *****/
179 1      PIDRUN: PROCEDURE INTERRUPT 3 PUBLIC;
      /* TURN THE LED INDICATOR ON */
180 2      OUTPUT (RESET$LATCH$ADR) = INDICATOR$ON;
      /* GET WEIGHBELT WEIGHT */
181 2      BELT$WEIGHT=WEIGHBELT$WEIGHT;
      /* GET LIQUID FLOW RATE */
182 2      LIQUID$FLOW=LIQUID$FLOW$RATE;
      /* CONTROL START-STOP RAMP */
183 2      IF SYSTEM$RUNNING
185 2      THEN MASS$SETPOINT=SETPOINT;
      ELSE MASS$SETPOINT=0;
      /* DETERMINE ACTUAL MASS FLOW ON WEIGHBELT */
186 2      CALL MQULD2(.IR,.BELT$CONTROL);
187 2      CALL MQUML2(.IR,.BELT$WEIGHT);
188 2      CALL MQUML1(.IR,.DIST$REV);
189 2      CALL MQUDV1(.IR,.CONV$LENGTH);
190 2      CALL MQSDV2(.IR,.THOUSAND);
191 2      CALL MQSST2(.IR,.MASS$FLOW);
      /* COMPUTE ERROR SIGNAL ON WEIGHBELT */
192 2      CALL MQSLD2(.IR,.MASS$SETPOINT);
193 2      CALL MQSSB2(.IR,.MASS$FLOW);
      /* HANDLE PID BELT CONTROL ALGORITHM */
194 2      CALL UQPPID(.PRCV,.IR);
      /* STORE OUTPUT SIGNAL */
195 2      CALL MQUST2(.IR,.BELT$CONTROL);

```

```

196 2 /* COMPUTE LIQUID SETPOINT */
197 2 CALL MQSLD2(.IR,.MASS$FLOW);
198 2 CALL MQSML1(.IR,.LIQUID$RATIO);
      CALL MQSML1(.IR,.SIX);

```

```

199 2 /* VERIFY THAT WEIGHBELT IS MOVING */
      IF WEIGHBELT$SPEED = 0
      THEN CALL MQULD2(.IR,.ZERO);
201 2 /* COMPUTE LIQUID ERROR */
      CALL MQSSB2(.IR,.LIQUID$FLOW);
202 2 /* HANDLE PID LIQUID CONTROL */
      CALL UQPPID(.PRLQ,.IR);
203 2 /* STORE OUTPUT SIGNAL */
      CALL MQUST1(.IR,.LIQUID$VALVE);
204 2 /* OUTPUT WEIGHBELT CONTROL SIGNAL */
      CALL WEIGHBELT$MOTOR$DRIVE (BELT$CONTROL);
205 2 /* OUTPUT FLOW CONTROL SIGNAL */
      CALL LIQUID$VALVE$POSITION (LIQUID$VALVE);
206 2 /* SEND END OF INTERRUPT TO 8259 CONTROLLER */
      OUTPUT(0ECH)=020H;
207 2 /* TURN THE LED INDICATOR OFF */
      OUTPUT (RESET$LATCH$ADR) = INDICATOR$OFF;
      /* RETURN FROM CONTROL TASK */
208 2 RETURN;
209 2 END PIDRUN;
210 1 END;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 01C1H      449D
VARIABLE AREA SIZE  = 0094H      148D
MAXIMUM STACK SIZE  = 000AH      10D
465 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE PROCESSORINITIALIZATIONMODULE
 OBJECT MODULE PLACED IN :F1:SBC941.OBJ
 COMPILER INVOKED BY: PLM80 :F1:SBC941.PLM DEBUG PAGEWIDTH(72) TITLE('PR
 OCESSOR INITIALIZATION')

```

/*****
* THIS PROGRAM IS USED TO INITIALIZE THE ISBC *
* 941 PROCESSOR INSTALLED IN SOCKET 0. THE *
* DEVICE WILL OPERATE IN THE FREQUENCY OUTPUT *
* MODE. *****/
*****

1      PROCESSOR$INITIALIZATION$MODULE: DO;

      /* DECLARATION OF ADDRESSES */
2      1      DECLARE UPI$0$STATUS LITERALLY '0E5H';
3      1      DECLARE UPI$0$COMMAND LITERALLY '0E5H';
4      1      DECLARE UPI$0$DATA LITERALLY '0E4H';

5      1      DECLARE UPI$1$STATUS LITERALLY '0E7H';
6      1      DECLARE UPI$1$COMMAND LITERALLY '0E7H';
7      1      DECLARE UPI$1$DATA LITERALLY '0E6H';

8      1      DECLARE UPI$2$STATUS LITERALLY '0E9H';
9      1      DECLARE UPI$2$COMMAND LITERALLY '0E9H';
10     1      DECLARE UPI$2$DATA LITERALLY '0E8H';

      /* DECLARATION OF ISBC 941 COMMANDS */
11     1      DECLARE SETP1 LITERALLY '00BH';
12     1      DECLARE CLRPI LITERALLY '00DH';
13     1      DECLARE CLRP2 LITERALLY '00EH';
14     1      DECLARE PAUSE LITERALLY '005H';
15     1      DECLARE LOOP LITERALLY '004H';
16     1      DECLARE INITPF LITERALLY '002H';
17     1      DECLARE PACIFY LITERALLY '001H';
18     1      DECLARE ENFLAG LITERALLY '006H';

      /* DECLARATION OF ISBC 941 STATUS BITS */
19     1      DECLARE RFC LITERALLY '080H';
20     1      DECLARE IBF LITERALLY '002H';
21     1      DECLARE QF LITERALLY '010H';

      /* DECLARATION OF ISBC 941 #0 INITIALIZATION DATA */
22     1      DECLARE FREQ LITERALLY '0B5H';
23     1      DECLARE SF LITERALLY '000H';
24     1      DECLARE OUTPUT$ENABLE0 LITERALLY '001H';
25     1      DECLARE INITIAL$STATE LITERALLY '000H';
26     1      DECLARE DELAY LITERALLY '001H';
27     1      DECLARE PERIOD LITERALLY '000H';
28     1      DECLARE INITIAL$OUTPUT LITERALLY '00EH';

```

```

/* DECLARATION OF INTERVAL TIMER PARAMETERS */
29 1 DECLARE PIT$0$MODE LITERALLY '016H';
30 1 DECLARE PIT$0$INTERVAL LITERALLY '00EH';
31 1 DECLARE PIT$0$MODE$WRD LITERALLY '0E3H';
32 1 DECLARE PIT$0$COUNT LITERALLY '0E0H';

/* DECLARATION OF COUNTER LOCATIONS */
33 1 DECLARE (LIQ$COUNT,BELT$COUNT) BYTE EXTERNAL;

/* DECLARATION OF ISBC 941 PRIMARY DATA */
34 1 DECLARE INIT$0$TABLE(6) BYTE DATA (
    FREQ,
    SF,
    OUTPUT$ENABLE0,
    INITIAL$STATE,
    DELAY,
    PERIOD );

/* DECLARATION OF MISC PARAMETERS */
35 1 DECLARE I BYTE;

*****
/* ISBC 941 INITIALIZATION PROGRAM BODY */
*****

36 1 ***** PROCESSOR$0$INITIALIZATION: PROCEDURE PUBLIC;

    /* INITIALIZE COUNTER 0 FOR 10 MICROSECONDS */
37 2 OUTPUT (PIT$0$MODE$WRD)=PIT$0$MODE;
38 2 OUTPUT (PIT$0$COUNT)=PIT$0$INTERVAL;

    /* VERIFY THAT PROCESSOR IS RESET */
39 2 DO WHILE ((INPUT(UPI$0$$STATUS) AND RFC) = 0);
40 3 DO WHILE ((INPUT(UPI$0$$STATUS) AND IBF) <> 0);
41 4 END;
42 3 OUTPUT (UPI$0$$COMMAND)=PACIFY;
43 3 END;

    /* REQUEST PRIMARY FUNCTION */
44 2 DO WHILE ((INPUT(UPI$0$$STATUS) AND IBF) <> 0);
45 3 END;
46 2 OUTPUT (UPI$0$$COMMAND)= INITPF;

*****/* LOAD INITIAL PARAMETERS */*****
47 2 DO I=0 TO 5;
48 3 DO WHILE ((INPUT(UPI$0$$STATUS) AND IBF) <> 0);
49 4 END;
50 3 OUTPUT (UPI$0$$DATA)=INIT$0$TABLE(I);
51 3 END;

    /* TERMINATE PARAMETER LOADING */
52 2 DO WHILE ((INPUT(UPI$0$$STATUS) AND IBF) <> 0);
53 3 END;
54 2 OUTPUT (UPI$0$$COMMAND)=PAUSE;

```

```

55 2  /* START FREQUENCY FUNCTION */
56 3  DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
57 2  END;
    OUTPUT(UPI$0$COMMAND)=LOOP;

    /* SET UNUSED BITS TO ALLOW EXPANSION */

58 2  DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
59 3  END;
60 2  OUTPUT(UPI$0$COMMAND)=CLRP2;

61 2  DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
62 3  END;
63 2  OUTPUT(UPI$0$DATA)=INITIAL$OUTPUT;

    /* RETURN TO CALLING PROGRAM */
64 2  RETURN;

65 2  END PROCESSOR$0$INITIALIZATION;
    $EJECT
    /* *****
    * THIS PROCEDURE IS USED TO INITIALIZE THE ISBC *
    * 941 PROCESSOR INSTALLED IN SOCKET 1. THE DE- *
    * VICE WILL OPERATE IN THE FCOUNT, HIGH FRE- *
    * QUENCY INPUT MODE. *
    * ***** */

    /* DEFINE INITIALIZATION PARAMETERS */
66 1  DECLARE FCOUNT LITERALLY '033H';
67 1  DECLARE INPUT$SELECT LITERALLY '001H';
68 1  DECLARE OUTPUT$ENABLE$1 LITERALLY '001H';
69 1  DECLARE SAMPLING$INTERVAL LITERALLY '009H';
70 1  DECLARE INITIAL$STATE$1 LITERALLY '0E1H';

    /* DECLARE PARAMETER INITIALIZATION TABLE */
71 1  DECLARE INIT$1$TABLE(4) BYTE DATA (
        FCOUNT,
        INPUT$SELECT,
        OUTPUT$ENABLE$1,
        SAMPLING$INTERVAL );

    /* *****
    * INITIALIZATION BODY *
    * ***** */

72 1  PROCESSOR$1$INITIALIZATION: PROCEDURE PUBLIC;

    /* VERIFY THAT PROCESSOR IS RESET */
73 2  DO WHILE ((INPUT(UPI$1$STATUS) AND RFC) = 0);
74 3  DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
75 4  END;
76 3  OUTPUT(UPI$1$COMMAND)=PACIFY;
77 3  END;

```



```

78 2 /* REQUEST PRIMARY FUNCTION */
79 3 DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
80 2 END;
      OUTPUT(UPI$1$COMMAND)=INITPF;

      /* LOAD INITIAL PARAMETERS */
81 2 DO I=0 TO 3;
82 3 DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
83 4 END;
84 3 OUTPUT(UPI$1$DATA)=INIT$1$TABLE(I);
85 3 END;

      /* TERMINATE PARAMETER LOADING */
86 2 DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
87 3 END;
88 2 OUTPUT(UPI$1$COMMAND)=PAUSE;

      /* SET UNUSED BITS HIGH FOR SPARE ENABLES */
89 2 DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
90 3 END;
91 2 OUTPUT(UPI$1$COMMAND)=SETP1;
92 2 DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
93 3 END;
94 2 OUTPUT(UPI$1$DATA)=INITIAL$STATE$1;

      /* START FREQUENCY COUNT OPERATION */
95 2 DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
96 3 END;
97 2 OUTPUT(UPI$1$COMMAND)=LOOP;

      /* RETURN TO CALLING PROGRAM */
98 2 RETURN;

99 2 END PROCESSOR$1$INITIALIZATION;

$EJECT
/*****
* THIS PROCEDURE IS USED TO INITIALIZE THE ISBC *
* 941 INSTALLED IN SOCKET 2. THE DEVICE WILL BE *
* OPERATED AS A STEPPER MOTOR DRIVER. *****/
*****
PROCESSOR$2$INITIALIZATION: PROCEDURE PUBLIC
/* DEFINE INITIALIZATION PARAMETERS */
100 1 DECLARE STEPPER$2$INITIAL$TABLE LITERALLY '017H';
101 1 DECLARE SCALE$FACTOR LITERALLY '0DFH';
102 1 DECLARE OUTPUT$ENABLE$2 LITERALLY '0F0H';
103 1 DECLARE OUTPUT$POLARITY LITERALLY '050H';
104 1 DECLARE COMMON$PERIOD LITERALLY '004H';
105 1 DECLARE P20$STRAN1 LITERALLY '000H';
106 1 DECLARE P20$STRAN2 LITERALLY '000H';
107 1 DECLARE P21$STRAN1 LITERALLY '000H';
108 1 DECLARE P21$STRAN2 LITERALLY '000H';
109 1 DECLARE P22$STRAN1 LITERALLY '000H';
110 1 DECLARE P22$STRAN2 LITERALLY '000H';

```

```

111 1 DECLARE P23$STRAN1 LITERALLY '000H';
112 1 DECLARE P23$STRAN2 LITERALLY '000H';
113 1 DECLARE P24$STRAN1 LITERALLY '000H';
114 1 DECLARE P24$STRAN2 LITERALLY '002H';
115 1 DECLARE P25$STRAN1 LITERALLY '000H';
116 1 DECLARE P25$STRAN2 LITERALLY '002H';
117 1 DECLARE P26$STRAN1 LITERALLY '001H';
118 1 DECLARE P26$STRAN2 LITERALLY '003H';
119 1 DECLARE P27$STRAN1 LITERALLY '001H';
120 1 DECLARE P27$STRAN2 LITERALLY '003H';

121 1 DECLARE CLR$LOW$PORT LITERALLY '00FH';

/* DECLARE PARAMETER INITIALIZATION TABLE */
122 1 DECLARE INIT$2$TABLE(21) BYTE DATA (
    STEPPER,
    SCALE$FACTOR,
    OUTPUT$ENABLES2,
    OUTPUT$POLARITY,
    COMMON$PERIOD,
    P20$STRAN1,
    P20$STRAN2,
    P21$STRAN1,
    P21$STRAN2,
    P22$STRAN1,
    P22$STRAN2,
    P23$STRAN1,
    P23$STRAN2,
    P24$STRAN1,
    P24$STRAN2,
    P25$STRAN1,
    P25$STRAN2,
    P26$STRAN1,
    P26$STRAN2,
    P27$STRAN1,
    P27$STRAN2);
/* ***** */
/* ***** INITIALIZATION BODY ***** */
/* ***** */

123 1 PROCESSOR$2$INITIALIZATION: PROCEDURE PUBLIC;
/* ***** */
/* ***** */
124 2 DO WHILE ((INPUT(UPI$2$STATUS) AND RFC) = 0);
125 3 DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
126 4 END;
127 3 OUTPUT(UPI$2$COMMAND)=PACIFY;
128 3 END;

/* REQUEST PRIMARY FUNCTION */
129 2 DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
130 3 END;
131 2 OUTPUT(UPI$2$COMMAND)=INITPF;

```

```

/* LOAD INITIAL PARAMETERS */
132 2 DO I=0 TO 20;
133 3 DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
134 * 4 END;
135 3 OUTPUT(UPI$2$DATA)=INIT$2$TABLE(I);
136 3 END;

/* TERMINATE PARAMETER LOADING */
137 2 DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
138 3 END;
139 2 OUTPUT(UPI$2$COMMAND)=PAUSE;

/* START STEPPER CONTROLLER OPERATION */
140 2 DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
141 3 END;
142 2 OUTPUT(UPI$2$COMMAND)=LOOP;

/* SET UNUSED BITS LOW TO ENABLE GENERAL FUNCTIONS */
143 2 DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
144 3 END;
145 2 OUTPUT(UPI$2$COMMAND)=CLR1;
146 2 DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
147 3 END;
148 2 OUTPUT(UPI$2$DATA)=CLR$LOW$PORT;

/* RETURN TO CALLING PROGRAM */
149 2 RETURN;

150 2 END PROCESSOR$2$INITIALIZATION;
$EJECT
/*****
* THIS PROCEDURE IS USED TO INITIALIZE COUNTER *
* 1 TO PERFORM AS AN EIGHT BIT BINARY COUNTER *
* WHICH WILL BE USED TO MEASURE THE BELT SPEED.*
*****/

151 1 COUNTER$1$INITIALIZATION: PROCEDURE PUBLIC;

/* INITIALIZE COUNTER 1 FOR EIGHT BIT COUNTING */
152 2 LIQ$COUNT = 0;

/* RETURN TO CALLING PROGRAM */
153 2 RETURN;

154 2 END COUNTER$1$INITIALIZATION;
$EJECT
/*****
* THIS PROCEDURE IS USED TO INITIALIZE COUNTER *
* 2 TO PERFORM AS AN EIGHT BIT BINARY COUNTER *
* WHICH WILL BE USED TO MEASURE THE LIQUID *
* FLOW THROUGH THE METER. *
*****/

```

```

155 1 COUNTER$2$INITIALIZATION: PROCEDURE PUBLIC;
/* INITIALIZE COUNTER 2 FOR EIGHT BIT COUNTING */
156 2 BELT$COUNT = 0;

/* RETURN TO CALLING PROGRAM */
157 2 RETURN;
END COUNTER$2$INITIALIZATION;
158 2
159 1 END PROCESSOR$INITIALIZATION$MODULE;
$SELECT

MODULE INFORMATION:
CODE AREA SIZE = 0201H 513D
VARIABLE AREA SIZE = 0001H 1D
MAXIMUM STACK SIZE = 0000H 0D
329 LINES READ
0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION

/* RETURN TO CALLING PROGRAM */
RETURN;

END PROCESSOR$2$INITIALIZATION;

$SELECT
*****
/* THIS PROCEDURE IS USED TO INITIALIZE COUNTER
* 1 TO PERFORM AS AN EIGHT BIT BINARY COUNTER
* WHICH WILL BE USED TO MEASURE THE BELT SPEED.
*****

COUNTER$1$INITIALIZATION: PROCEDURE PUBLIC;

/* INITIALIZE COUNTER 1 FOR EIGHT BIT COUNTING */
LIP$COUNT = 0;

/* RETURN TO CALLING PROGRAM */
RETURN;

END COUNTER$1$INITIALIZATION;

$SELECT
*****
/* THIS PROCEDURE IS USED TO INITIALIZE COUNTER
* 2 TO PERFORM AS AN EIGHT BIT BINARY COUNTER
* WHICH WILL BE USED TO MEASURE THE LIQUID
* FLOW THROUGH THE METER.
*****

```

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE PROCESSORINTERFACEMODULE
 OBJECT MODULE PLACED IN :F1:OPR941.OBJ
 COMPILER INVOKED BY: PLM80 :F1:OPR941.PLM DEBUG

```

$INTVECTOR(4,3F00H)
$PAGEWIDTH(72)
$TITLE('PROCESSOR INTERFACE')
/*****
 * THESE PROGRAMS PROVIDE THE OPERATING INTER-
 * FACE BETWEEN THE APPLICATION PROGRAM AND
 * THE ISBC 941 PROCESSORS OR COUNTERS.
 *****/

1      PROCESSOR$INTERFACE$MODULE: DO;
      /*
      /* DECLARATION OF ADDRESSES */
2      1 DECLARE UPI$0$STATUS LITERALLY '0E5H';
3      1 DECLARE UPI$0$COMMAND LITERALLY '0E5H';
4      1 DECLARE UPI$0$DATA LITERALLY '0E4H';
5      1 DECLARE UPI$1$STATUS LITERALLY '0E7H';
6      1 DECLARE UPI$1$COMMAND LITERALLY '0E7H';
7      1 DECLARE UPI$1$DATA LITERALLY '0E6H';
8      1 DECLARE UPI$2$STATUS LITERALLY '0E9H';
9      1 DECLARE UPI$2$COMMAND LITERALLY '0E9H';
10     1 DECLARE UPI$2$DATA LITERALLY '0E8H';
      /*
      /* DECLARATION OF ISBC 941 COMMANDS */
11     1 DECLARE SETP1 LITERALLY '00BH';
12     1 DECLARE CLRP1 LITERALLY '00DH';
13     1 DECLARE CLRP2 LITERALLY '00EH';
14     1 DECLARE RDFC0 LITERALLY '042H';
15     1 DECLARE RDFC1 LITERALLY '043H';
16     1 DECLARE PAUSE LITERALLY '005H';
17     1 DECLARE LOOP LITERALLY '004H';
18     1 DECLARE INITPF LITERALLY '002H';
19     1 DECLARE PACIFY LITERALLY '001H';
20     1 DECLARE ENFLAG LITERALLY '006H';
21     1 DECLARE SETOE LITERALLY '071H';
      /*
      /* DECLARATION OF ISBC 941 STATUS BITS */
22     1 DECLARE RFC LITERALLY '080H';
23     1 DECLARE IBF LITERALLY '002H';
24     1 DECLARE OBF LITERALLY '001H';
25     1 DECLARE QF LITERALLY '010H';
26     1 DECLARE QNE LITERALLY '020H';
      /*
      /* DEFINE THE MATH ROUTINES USED BY MODULES */
27     1 MQULD4: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
28     2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
29     2 END MQULD4;
30     1 MQUDV2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
31     2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
32     2 END MQUDV2;

```



```

33 1 MQUDV1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;--
34 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
35 2 END MQUDV1;
36 1 MQUST1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
37 2 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
38 2 END MQUST1;

/* DEFINE THE MATH ACCUMULATOR STORAGE AREA */
39 1 DECLARE IR(18) BYTE EXTERNAL;

/* DEFINE THE COUNTER LOCATIONS */
40 1 DECLARE (LIQ$COUNT,BELT$COUNT) BYTE EXTERNAL;

$EJECT
/*****
* THIS PROGRAM IS USED TO GENERATE A FREQUENCY *
* OUTPUT USING THE ISBC 941 MODULE INSTALLED IN *
* SOCKET NUMBER 0. TO PROVIDE MAXIMUM RESOLU- *
* TION, FOUR PERIODS WILL BE USED. THE FREQUEN- *
* CY RANGES CORRESPONDING TO EACH PERIOD ARE: *
* RANGE FREQ RESOLUTION *
* 1 50 TO 165 HZ 2 HZ *
* 2 166 TO 225 HZ 3 HZ *
* 3 226 TO 285 HZ 3 HZ *
* 4 286 TO 550 HZ 6 HZ *
* THE SCALE FACTOR IS COMPUTED BY THE FORMULA: *
* SF=100000/((FREQ)*(RANGE FACTOR)) *
*****/

41 1 WEIGHBELT$MOTOR$DRIVE: PROCEDURE (FREQ) PUBLIC;

/* DECLARATION OF CONSTANT, 100,000 */
42 2 DECLARE HUNDRED$K(4) BYTE DATA (
0A0H,086H,001H,000H );

/* DECLARATION OF ISBC941 PORT ENABLES */
43 2 DECLARE ENABLE$FREQ LITERALLY '01H';
44 2 DECLARE DISABLE$FREQ LITERALLY '00H';

/* DECLARATION OF ISBC 941 MEMORY LOCATION COMMANDS */
45 2 DECLARE WRRM$55 LITERALLY '055H';
46 2 DECLARE WRRM$74 LITERALLY '074H';

/* DECLARATION OF VARIABLES USED IN COMPUTATIONS */
47 2 DECLARE (RANGE,FREQA) BYTE;
48 2 DECLARE FREQ ADDRESS;

/* BEGIN COMPUTATION OF OUTPUT FOR FREQ > 48 HZ. */
49 2 IF FREQ > 48
THEN DO;

/* ENABLE FREQUENCY OUTPUT */
51 3 DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
52 4 END;
53 3 OUTPUT(UPI$0$COMMAND) = SETOE;

```

```

54 3 DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
55 4 END;
56 3 OUTPUT(UPI$0$DATA) = ENABLE$FREQ;

/* COMPUTATION OF FREQUENCY RANGE */
57 3 IF FREQ < 285
    THEN DO;
59 4 IF FREQ < 226
    THEN DO;
61 5 IF FREQ < 166
    THEN RANGE = 9;
    ELSE RANGE = 5;
63 5 END;
64 5 ELSE RANGE = 3;
65 4 END;
66 4 ELSE RANGE = 2;
67 3

/* LOAD MATH ACCUMULATOR WITH 100,000 */
68 3 CALL MQULD4 (.IR,.HUNDRED$K);

/* TEST FOR MOTOR SHUTDOWN */
69 3 IF FREQ > 1
    THEN DO;

/* DIVIDE BY FREQUENCY */
71 4 CALL MQUDV2 (.IR,.FREQ);

/* DIVIDE BY RNAGE FACTOR */
72 4 CALL MQUDV1 (.IR,.RANGE);

*****/* GET TWO'S COMPLEMENT FOR ISBC 941 SCALE FACTOR */
73 4 CALL MQUST1 (.IR,.FREQA);
74 4 FREQA = NOT (FREQA + 1);
75 4 END;

/* ADJUST FOR MOTOR STOP SIGNAL */
76 3 ELSE DO;
77 4 FREQA = 000H;
78 4 RANGE = 0FFH;
79 4 END;

/* SEND NEW SCALE FACTOR TO DEVICE */
80 3 DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
81 4 END;
82 3 OUTPUT(UPI$0$COMMAND) = WRRM$55;

83 3 DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
84 4 END;
85 3 OUTPUT(UPI$0$DATA) = FREQA;

/* SEND NEW PERIOD TO DEVICE */
86 3 DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
87 4 END;
88 3 OUTPUT(UPI$0$COMMAND) = WRRM$74;

```

```

89      3      DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
90      4      END;
91      3      OUTPUT(UPI$0$DATA) = RANGE;

/* END OF FREQUENCY OUTPUT MODE */
92      3      END;

/* HANDLE FREQUENCIES < 50 HZ. */
93      2      ELSE DO;

/* DISABLE FREQUENCY OUTPUT GENERATION */
94      3      DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
95      4      END;
96      3      OUTPUT(UPI$0$COMMAND) = SETOE;

97      3      DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
98      4      END;
99      3      OUTPUT(UPI$0$DATA) = DISABLE$FREQ;

/* END OF ALTERNATE FREQUENCY OUTPUT */
100     3      END;

/* RETURN TO CALLING PROGRAM */
101     2      RETURN;

102     2      END WEIGHBELT$MOTOR$DRIVE;

$EJECT
/*****
* THIS PROGRAM GETS THE WEIGHBELT WEIGHT FROM THE
* NUMBER 1 ISBC 941 PROCESSOR. THE WEIGHT WILL BE
* RECEIVED AS A COUNT WHICH RANGES BETWEEN 0 AND
* 2000, CORRESPONDING TO A WEIGHT BETWEEN 0.0 AND
* 10.00 POUNDS. EACH COUNT RECEIVED HAS A VALUE
* OF 0.005 POUNDS.
*****/
103     1      WEIGHBELT$WEIGHT: PROCEDURE ADDRESS PUBLIC;

/* DECLARATIONS OF VARIABLES USED IN THE PROCEDURE */
104     2      DECLARE (LCOUNT,HCOUNT) BYTE;
105     2      DECLARE WEIGHT ADDRESS;

/* GET INPUT COUNT LOW BYTE */
106     2      DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
107     3      END;
108     2      OUTPUT(UPI$1$COMMAND) = RDFO;

109     2      DO WHILE ((INPUT(UPI$1$STATUS) AND OBF) = 0);
110     3      END;
111     2      LCOUNT = INPUT(UPI$1$DATA);

```

```

*****
112 2  /* GET INPUT COUNT HIGH BYTE */
113 3  DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
114 2  END;
      OUTPUT(UPI$1$COMMAND) = RDPFC1;
*
115 2  DO WHILE ((INPUT(UPI$1$STATUS) AND OBF) = 0);
116 3  END;
117 2  HCOUNT = INPUT(UPI$1$DATA);
*****
      /* START NEXT WEIGHT SAMPLE PERIOD */
118 2  DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
119 3  END;
120 2  OUTPUT(UPI$1$COMMAND) = LOOP;
      /* CONVERT WEIGHT TO AN ADDRESS VALUE */
121 2  WEIGHT = HCOUNT;
122 2  WEIGHT = SHL(WEIGHT,8);
123 2  WEIGHT = WEIGHT + LCOUNT;
      /* DIVIDE BY TWO TO CONVERT TO POUNDS */
124 2  WEIGHT = SHR(WEIGHT,1);
      /* RETURN THE WEIGHTBELT WEIGHT */
125 2  RETURN WEIGHT;
126 2  END WEIGHBELT$WEIGHT;
$EJECT
/*****
* THIS PROCEDURE TRANSFERS THE WEIGHBELT SPEED TO
* THE CALLING PROGRAM AND CLEARS THE COUNTER FOR
* THE NEXT TEST. THE SPEED RESOLUTION PROVIDES
* ONLY FIVE SPEED RANGES.
*****/
127 1  WEIGHBELT$SPEED: PROCEDURE BYTE PUBLIC;
      /* DECLARATIONS OF VARIABLES USED BY THE PROCEDURE */
128 2  DECLARE SPEED BYTE;
      /* LATCH COUNTER BEFORE READING SPEED */
129 2  DISABLE;
      /* GET COUNTER VALUE FROM COUNTER */
130 2  SPEED = BELT$COUNT;
      /* CLEAR COUNTER FOR NEXT OPERATION */
131 2  BELT$COUNT = 0;
132 2  ENABLE;
      /* RETURN DATA TO CALLING ROUTINE */
133 2  RETURN SPEED;
134 2  END WEIGHBELT$SPEED;

```

```

* PUTE THE SIGNED MAGNITUDE REPRESENTATION FROM *
* THE TWO'S COMPLIMENT INPUT AND WILL ISSUE THE *
* APPROPRIATE STEP INCREMENT AND DIRECTION. A *
* FIXED STEP RATE OF 100 STEPS PER SECOND WILL BE *
* USED BY THE CONTROL DEVICE. *
*****/
135 1 <> LIQUID$VALVE$POSITION: PROCEDURE (POSITION) PUBLIC;

/* DECLARATIONS OF VARIABLES USED BY THE PROCEDURE */
136 2 DECLARE POSITION BYTE;

/* DEFINITIONS OF TERMS USED IN COMPUTATIONS */
137 2 DECLARE STEP$RATE LITERALLY '005H';
138 2 DECLARE REVERSE LITERALLY '080H';

/* IF NO MOVEMENT, SKIP OPERATIONS */
139 2 IF POSITION <> 0
    THEN DO;

/* SUPPORT CONVERSION TO SIGNED MAGNITUDE NUMBER */
141 3 IF POSITION > 127
    THEN DO;

/* GET MAGNITUDE OF MOVEMENT */
143 4 POSITION = 256 - POSITION;

/* SET SIGN FOR CCW ROTATION */
144 4 POSITION = POSITION OR REVERSE;
145 4 END;

/* VERIFY THAT QUEUE SPACE IS AVAILABLE */
146 3 DO WHILE ((INPUT(UPI$2$STATUS) AND QF) <> 0);
147 4 END;

/* REQUEST DESIRED STEP RATE */
148 3 DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
149 4 END;
150 3 OUTPUT(UPI$2$DATA) = STEP$RATE;

/* REQUEST STEPPER MOVEMENT */
151 3 DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
152 4 END;
153 3 OUTPUT(UPI$2$DATA) = POSITION;
154 3 END;

/* RETURN TO CALLING PROGRAM */
155 2 RETURN;

156 2 END LIQUID$VALVE$POSITION;

```



```

$EJECT
/*****
* THIS PROCEDURE TRANSFERS THE LIQUID FLOW RATE FROM *
* THE FLOW COUNTER TO THE CALLING PROGRAM. AFTER *
* READING, THE FLOW COUNTER WILL BE RESET TO FACILI- *
* TATE THE NEXT READING. THE LIQUID FLOW COUNT WILL *
* VARY BETWEEN 20 AND 240 PULSES IN EACH 200 MILLI- *
* SECOND SAMPLE INTERVAL. THIS WILL CORRESPOND TO *
* THE ACTUAL LIQUID FLOW RATE OF 10 TO 120 POUNDS *
* PER MINUTE. *
*****/

157 1 LIQUID$FLOW$RATE: PROCEDURE ADDRESS PUBLIC;

/* DECLARATION OF VARIABLES USED BY THE PROGRAM */
158 2 DECLARE TEMP BYTE;
159 2 DECLARE (FLOW,T$TWO,T$SXTN,T$THRTWO) ADDRESS;

/* LATCH COUNTER BEFORE READING FLOW */
160 2 DISABLE;

/* GET FLOW RATE VALUE FROM COUNTER */
161 2 TEMP = LIQ$COUNT;

/* CLEAR COUNTER FOR NEXT OPERATION */
162 2 LIQ$COUNT = 0;
163 2 ENABLE;

/* CONVERT TO POUNDS PER MINUTE */
164 2 FLOW = TEMP;
165 2 T$TWO = SHL(FLOW,1);
166 2 T$SXTN = SHL(T$TWO,3);
167 2 T$THRTWO = SHL(T$SXTN,1);
168 2 FLOW = T$TWO + T$SXTN + T$THRTWO;

/* RETURN FLOW RATE TO CALLING PROGRAM */
169 2 RETURN FLOW;

170 2 END LIQUID$FLOW$RATE;

$EJECT
/*****
* COUNTER FOR LIQUID FLOW RATE FROM LIQUID *
* FLOW METER. COUNT PULSE WILL GENERATE AN *
* INTERRUPT AT LEVEL 1. *
*****/

171 1 LIQ$CNT: PROCEDURE INTERRUPT 1 PUBLIC;

/* INCREMENT FLOW COUNT */
172 2 LIQ$COUNT = LIQ$COUNT + 1;

/* SEND END OF INTERRUPT */
173 2 OUTPUT (0ECH) = 020H;

```

```

SELECT
*****
/* RETURN */
*****
174 2 RETURN;
* THIS PROCEDURE TRANSFERS THE LIQUID FLOW RATE FROM
* THE FLOW COUNTER TO THE CALLING PROGRAM. AFTER
* READING, THE FLOW COUNTER WILL
* STATE THE NEXT READING. THE FLOW COUNTER WILL
* VARY BETWEEN 0 AND 128 MILLI-
* LITERS PER HOUR. EACH 256 MILLI-
* LITERS CORRESPOND TO
* 128 POUNDS
*
* THIS PROCEDURE WILL PROVIDE AN EVENT COUN-
* TER TO HANDLE THE BELT MOTION DETECTOR.
* IT WILL OPERATE BY DIRECTING THE MOTION
* PULSE TO INTERRUPT 2.
*****
175 2 $EJECT
/* *****
* THIS PROCEDURE WILL PROVIDE AN EVENT COUN-
* TER TO HANDLE THE BELT MOTION DETECTOR.
* IT WILL OPERATE BY DIRECTING THE MOTION
* PULSE TO INTERRUPT 2.
*****
176 1 BELT$CNT: PROCEDURE INTERRUPT 0 PUBLIC;
/* INCREMENT BELT MOVEMENT */
177 2 BELT$COUNT = BELT$COUNT + 1;
/* SEND END OF INTERRUPT */
178 2 OUTPUT (0ECH) = 020H;
/* RETURN */
179 2 RETURN;
/* CLEAR COUNTER FOR NEXT OPERATION */
180 2 END BELT$CNT;
181 1 END PROCESSOR$INTERFACE$MODULE;
/* CONVERT TO POUNDS PER MINUTE */
/* FLOW = TEMP;
/* TSTWO = SHL(FLOW,1);
/* TSTXIN = SHL(TSTWO,1);
/* TSTRTW = SHL(TSTXIN,1);
/* FLOW = TSTRTW + TSTWO + 8D
/* RETURN FLOW RATE TO CALLING
/* RETURN FLOW;
END LIQUID$FLOWRATE;
SELECT
*****
/* COUNTER FOR LIQUID FLOW RATE FROM LIQUID
/* FLOW METER. COUNT PULSE WILL GENERATE AN
/* INTERRUPT AT LEVEL 1.
*****
LIQ$CNT: PROCEDURE INTERRUPT 1 PUBLIC;
/* INCREMENT FLOW COUNT */
LIQ$COUNT = LIQ$COUNT + 1;
/* SEND END OF INTERRUPT */
OUTPUT (RECH) = 020H;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0251H 593D
VARIABLE AREA SIZE  = 0013H 19D
MAXIMUM STACK SIZE  = 0008H 8D
400 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

■ ■ ■ ■ ■

... ..

© 2000 Blackwell Science Ltd *Journal of Internal Medicine* 247: 111–117

By Michael J. McGowan
Control Engineering

19

Designing and Assembling Microcomputer Systems Grows Easier

Although a single data bus standard yet eludes the microcomputer industry, numerous manufacturers of single-board computer and supplementary boards have cast a hardware vote for the Multibus, Intel's microcomputer backplane which they originated in 1976. With a steady eye on the control industry market, Intel has designed a home to accommodate Multibus compatible equipment, the iCS-80 industrial chassis. It promises to significantly reduce the time and cost of assembling the housing and interface parts of a microcomputer-based control system. In this article, besides taking the first look at Intel's new chassis and signal conditioning panels, we've put together a comprehensive list of Multibus compatible equipment.

MICHAEL J. MCGOWAN, Control Engineering

After the development of single-board computers nearly three years ago, vendors moved quickly to seize a fraction of the market. It seemed at first that everything from memories to analog I/O boards had become available. With an astonishing suddenness, companies sprang up in Silicon Valley, Texas, New Jersey, and along the forested roadside of Rt. 128 outside of Boston. Late that year, we counted well over a hundred companies anxious to make their fortune selling the control engineer everything from one or two interface boards to complete microprocessor systems.

Then the pleasant dream became a nightmare. From power supply requirements to backplane pinouts, little was compatible. Even such an obvious thing as board size differed from vendor to vendor and many a hope for an ideal system was crushed in a pragmatic search for whatever would fit together.

Seven months ago in our June 1978 issue we noted that the number of microcomputer system manufacturers had dwindled to about 60 and since then, we find still fewer. Some, no doubt, were forced out for lack of reliability, though most, despite remarkably talented engineering, starved as the market saturated.

Large scale integration of microcomputer components has more than doubled the memory size of single-board computers. Sixteen bit word lengths will become commonplace in the next year as microcomputer performance begins to rival the minicomputer's and the lines of distinction between micros and minis fades.

The fight for a standard data bus drags on with leaders in the struggle but no winner. On the offensive, Pro-Log and Mostek jointly introduced the STD bus last autumn in an attempt to gain a greater market share by espousing decentralized system architectures. Their philosophy argues economics: the user should pay only for essential functions by selecting small, specialized boards

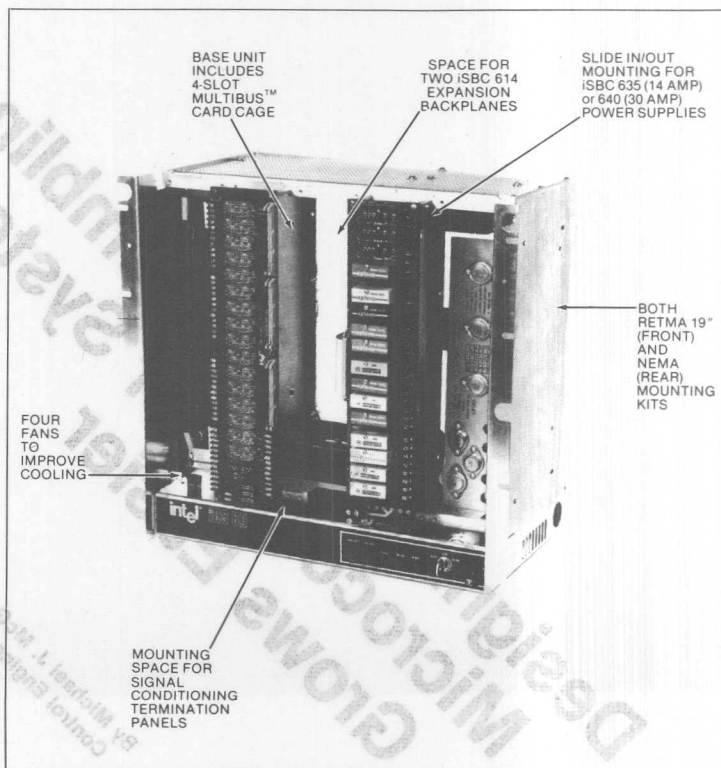
and not squander funds on a general purpose board with extra features.

Still, Intel, favoring more densely packed and versatile boards, continues to dominate the market while the Multibus retains its popularity. Today more 30 manufacturers produce over one hundred different boards based on that bus structure alone. Not that Intel enjoys the strict fidelity of its outside vendors; Digital Equipment Corporation, for instance, boasts some 17 companies providing boards to mate with the LSI-11 and LSI-112.

But perhaps alone among its competitors, Intel has recognized that the majority of its boards are being used in industrial applications and that the control system designer needs more than components.

An industrial chassis

A microcomputer system designer must choose components that are electro-mechanically compatible. To that end, Intel is introducing the iCS-80 industrial chassis and termination panels. It makes all Multibus-compatible CPU



MULTIBUS Compatible Boards and Vendors	Analog Input and Output Boards	Analog Signal Conditioning/ Screw Terminations	Communications Boards	Core Memory Boards	Digital Input/Output Boards	Digital Relay Boards	CPU Boards	DMA Controller Boards	Floppy Disk Controller Boards	Hard Disk Interface Boards	IEEE488 Bus Interface Boards	Keyboard Controller Boards	Math Boards	Optical Isolator Boards (without screw terminations)	Optical Isolation With Screw Terminations	PROM Boards	RAM Boards	RAM/PROM or ROM	Synchro to Digital Boards	Tape (Cartridge/cassette) Interface Boards	Tape (1/2") Interface Boards	Video Boards	Wire Wrap Boards
ADAC Corp.																							
Advanced Micro Computers																							
Ampex																							
Analog Devices																							
Augat																							
Burr-Brown																							
Computer Marketing																							
Data Translation																							
Datacube																							
Datel Systems																							
Electronic Solutions																							
Garry Manufacturing																							
HT Instruments																							
Hal Communications																							
Heurikon																							
IDEAS																							
Intel																							
Interphase																							
Matrox Electronic Systems																							
Megalogic																							
Micro Memories																							
Micro Networks																							
MicroTec																							
Micro/Tel																							
Monolithic Systems																							
Motorola																							
MUPRO																							
National Semiconductor																							
North Star Computers																							
Pertec (ICOM)																							
Relational Memory Systems																							
Systems, Computers and Interfaces																							
Thomas Engineering Co.																							
Vector Electronic																							
XEDAX																							
ZIA Tech																							

and peripheral boards readily usable.

The advantage of the ICS-80 is that most of the interconnection and mechanical details for assembling a microcomputer-based control system have already been worked out.

The iCS-80 stands 15.75 inches high and can be mounted in a standard RETMA 19 inch rack, or in a NEMA cabinet secure from the industrial environment. The minimum layout consists of a four-slot Multibus card cage with provisions for adding two more cages to a maximum of twelve cards. The cages fit vertically, like records in a rack, to aid convection cooling and permit front access for insertion and maintenance.

On the right side of the chassis is room for either a 14 or 30 ampere power supply, the choice dictated by the application. The system will operate on either 115 or 230 volts with a range of 47 to 63 Hertz specified in anticipation of international service.

Cooling is assisted by four fans—three for the card cages and one for the power supply section. The intention here is to make the installation of additional fans unnecessary even after the system has expanded. The fans are expected to provide adequate cooling for most applications so supplementary air conditioning can be eliminated or at least minimized.

Signal conditioning

Three signal conditioning panels have been developed by Intel to simplify connections between the processing cards and the outside world. The principle is neatness, and with that follows reliability. Flat ribbon cables connect the signal conditioners to the processor cards, a safeguard from "which wire is which" and screwdriver slips in the vicinity of expensive boards. Field connections to the external inputs and outputs are made (presumably by electricians with big hands and reputations for being less than delicate) through rugged, screw-type barrier strips that accept wire as heavy as 14 AWG. The panels can mount either on RETMA cabinet brackets, NEMA wall spacers, or on the ICS-80 chassis itself.

Each signal conditioning card gives the user a variety of options. The iCS-910 analog signal conditioning/termination panel accepts up to 16 differential or 32 single ended input channels. The four 2-wire analog output channels might be connected to 4 to 20 mA current loops.

The digital signal conditioning termination panel, iCS-920, handles 24 two-wire input or output channels with signals up to 55 V, 300 mA. Inputs can be diode protected, and pads are provided for current limiters or voltage dividers. Optoisolators may be inserted in

Vendors of Multicompatible Boards

ADAC Corp.
Woburn, MA
617/935-6668

Advanced Micro Computers
Santa Clara, CA
408/732-2400

Ampex
El Segundo, CA
714/973-2970

Analog Devices
Norwood, MA
617/329-4700

Augat
Attleboro, MA
617/222-2202

Burr-Brown
Tucson, AZ
602/655-8000

Computer Marketing
Waltham, MA
617/894-7000

Data Translation
Natick, MA
617/655-5300

Datcube
Reading, MA
617/944-4600

Datel Systems
Canton, MA
617/828-8000

Electronic Solutions
San Diego, CA
714/292-0242

Garry Manufacturing
New Brunswick, NJ
212/267-6844

HT Instruments
Marina Del Rey, CA
312/822-4296

Hal Communications
Urbana, IL
217/367-7373

Heurikon
Madison, WI
608/255-9075

IDEAS
Beltsville, MD
301/937-3600

Intel
Aloha, OR
503/642-2563

Interphase
Dallas, TX
214/238-0971

Matrox Electronic Systems
Montreal, Quebec
514/735-1182

Megalogic
Brookville, OH
513/833-5222

Micro Memories
Chatsworth, CA
213/998-0070

Micro Networks
Worcester, MA
617/852-5400

MicroTec
Sunnyvale, CA
408/733-2919

Micro/Tel
St. Louis, MO
314/569-3450

Monolithic Systems
Englewood, CO
303/770-7400

Motorola
Austin, TX
512/928-6572

MUPRO
Sunnyvale, CA
408/737-0500

National Semiconductor
Santa Clara, CA
408/737-5262

North Star Computers
Berkeley, CA
415/549-0858

Pertec (ICOM)
Chatsworth, CA
213/998-1800

Relational Memory Systems
San Jose, CA
408/248-6356

Systems, Computers and Interfaces
Waltham, MA
617/899-2359

Thomas Engineering Co.
Concord, CA
415/686-3041

Vector Electronic
Sylmar, CA
213/365-9661

XEDAX
Alameda, CA
415/521-6600

ZIA Tech
Cupertino, CA
408/996-7082

the DIP sockets for high voltage isolation or jumpers may be used instead when the input is TTL. Similarly, output sockets accept jumpers for direct TTL output, DIP optoisolators for transient suppression, or integrated circuit (open collector) drivers for high voltage to high current outputs. Activity on each channel is indicated by LEDs.

The ac signal conditioning/solidas termination panel, iCS-930, will actually work with ac or dc on its 16 channels. The user supplies optoisolators for input isolation and optically-isolated solid

state relays for output isolation. Mounting pads for customer-supplied MOVs or snubber networks are included. As before, a fuse gives overload protection and LEDs indicate channel activity.

The advantage of all this is that by plugging in some components and perhaps inserting a few resistors and capacitors, the interface units can be tailored to a particular application. Since many mechanical and electrical connection problems have already been solved, a customized unit can be built with minimum effort. □

DOCUMENTATION

RELATED INTEL PUBLICATIONS

System 80/10 Microcomputer Hardware Reference Manual, 98-00316B

iSBC 80/10 and iSBC 80/10A Single Board Computer Hardware Reference Manual, 9800230F

iSBC 80P and iSBC 80P10 Prototyping Package User's Guide, 9800223D

iSBC 80/20 and iSBC 80/20-4 Single Board Computer Hardware Reference Manual, 98-317C

iSBC 80/30 Hardware Reference Manual, 9800611A

iSBC 86/12 Single Board Computer Hardware Reference Manual, 9800645A

iSBC 544 Intelligent Communications Controller Board Hardware Reference Manual, 9800616B

iSBC 569 Intelligent Digital Controller Board Hardware Reference Manual, 9800845

iSBC 941 Industrial Digital Processor User's Guide, 9803077-02

iCS 80 Industrial Chassis Hardware Reference Manual, 9800799A

iSBC 310 High Speed Mathematics Unit Hardware Reference Manual, 9800410A

iSBC 957 Intellec-iSBC 86/12 Interface and Execution Package User's Guide, 9800743A

Intel MULTIBUS Specification, 9800683

MCS-80 User's Manual, 98-153D

MCS-85 User's Manual, 98-366C

The 8086 Family User's Manual, 9800722

UPI-41 User's Manual, 9800504

Introduction to the UPI-41A, AP-41

RMX/80 User's Guide, 9800522C

ISIS-II User's Guide, 9800306D

8080/8085 Assembly Language Programming Manual, 9800301C

PL/M-80 Programming Manual, 9800268B

ISIS-II PL/M-80 Compiler Operator's Manual, 9800300

FORTTRAN-80 Programming Manual, 9800481A

ISIS-II FORTRAN-80 Compiler Operator's Manual, 9800480B

"How to use FORTRAN with other Intel Languages", AP-44

BASIC-80 Reference Manual, 9800758

A Guide to Intellec Microcomputer Development Systems by Daniel D. McCracken, 9800558B

8080/8085 Fundamental Support Package (FSP) Reference and Operating Instructions for ISIS-II Users, 9800887-01

8086 Assembly Language Reference Manual, 9800640A

MCS-86 Assembler Operator's Instructions for ISIS-II Users, 9800641A

PL/M-86 Programming Manual, 9800466A

ISIS-II PL/M-86 Compiler Operator's Manual, 9800478A

ISIS-II 8086 Cross Development Utilities Operator's Manual, 9800639A

Intel iSBX Bus Specification, 142686-001

iSBC 80/10B Single Board Computer Hardware Reference Manual, 9803119-01

iSBC 80/24 Single Board Computer Hardware Reference Manual, 142648-001

iSBX 350 Parallel MULTIMODULE Board Hardware Reference Manual, 9803191-01

iSBX 351 Serial MULTIMODULE Board Hardware Reference Manual, 9803190-01

iSBX 331 Fixed/Floating Point Math MULTIMODULE Board Hardware Reference Manual, 142668-001

Introduction to the iRMX 86 Operating System, 9803124

iRMX 86 Nucleus, Terminal Handler, and Debugger Reference Manual, 9803122

iRMX 86 I/O System Reference Manual, 9803123

iRMX 86 System Programmer's Reference Manual, 142721

iRMX 86 Installation Guide for ISIS-II Users, 9803125

iRMX 86 Configuration Guide for ISIS-II Users, 9803126

iSBC 534 Four Channel Communications Expansion Board Hardware Reference Manual, 9800450

RMX/80 Interactive Configuration Utility User's Guide, 142603

Simplifying Complex Designs Using the iRMX 80 Nucleus, AP-112, 143349

iSBC 86/12A Single Board Computer Hardware Reference Manual, 9803074

8086 Family Utilities User's Guide, 980639

iSBC 88/40 Hardware Reference Manual

TECHNICAL LITERATURE LIST

Title	Part No.
MEMORY COMPONENTS	
Memory Design Handbook — 1979	011100
Growing Static RAM Family Album	010100
2115A/2125A Brochure	001710
RR 7 — 2107A/2107B Reliability	006540
RR 8 — Polysilicon fuse Bipolar PROM	006560
RR 11 — 2416 16K CCD Memory	006700
RR 12 — 2708 8K Erasable PROM	006720
RR 14 — 2115/2125 MOS Static RAMS	006740
RR 15 — 2104A	006750
RR 16 — 2116	006760
RR 18 — HMOS Reliability Update	006771
RR 19-2716 — UV Erasable PROM	006775
RR 20-2117 — Reliability	006780
AR 20 — 16K RAM	006900
AR 35-2716 — Erasable PROM-16,384 Bits On-Chip	007300
AR 44 — Speedy RAM Runs Cool — 2147	007320
AR 46 — HMOS Scales Traditional Devices	007330
AR 78 — ISSCC Reprint on Static RAMS	007370
AP 22 — Which Way for 16K	008300
AP 23 — 2104A 4K RAM	008500
AP 30 — Applications of 5 Volt EPROM & ROM Family	008550
AP 46 — Error Detecting and Correcting Codes	008560
TELECOM	
AR 79 — ISSCC Reprint — 2920	007375
AR 80 — ISSCC Reprint — 2912	007380
AR 81 — Single Chip NMOS Micro-process Signals	007385
AR 88 — First Monolithic PCM Filter	007400
MAGNETICS	
Bubble Memory Design Handbook	900020
AR 92 — Megabit Bubble Memory Chip Gets Support from LSI	900500
AR 96 — Here Comes A Million Bit Chip	900515
A Total System Solution to Magnetics Applications (Technical Paper)	900520
MICROCOMPUTER COMPONENTS	
MCS 48 User's Manual	98-270
MCS 48 Product Description (98-615)	201710
MCS 48 Applications Handbook	121511
MCS 48 Reference Card (98-412)	202300
AP 24 — MCS 48 Family (98-413)	203800
AP 40 — Keyboard/Display Scanning . . . MCS 48 (98-755)	203805
AP 49 — Serial I/O and Math Utilities . . . 8049 (98-904)	203810
AP 55A — High Speed Emulator for MCS 48	203815
AP 56 — Designing With Intel's 8022 Micro (98-954)	203820
AR 58 — Microcontroller Includes A-D Converter (98-718)	203605
AR 63 — Microcomputer's On-Chip Functions — 8022 (98-780)	203610
AR 102 — Designing Reliable Software for Auto Applications	207350
AR 107 — Use EPROM 1-Chip μ Cs as Effective 1-Shot Lab Aids	207355
UPI-41 User's Manual	98-504
UPI-41 Reference Card (98-671)	203100
MCS-48 and UPI-41 Assembly Language Programming Manual	98-255
MCS-80 User's Manual	98-153
RR 10 — 8080/8080A Microcomputer	207100
MCS-85 User's Manual	98-366
MCS-85 Product Description (98-365)	205770
8080/8085 Reference Card (98-438)	205785
AP 29 — Using the Intel 8085 Serial I/O Lines (98-684)	207715
8080/8085 Assembly Language Programming Manual	98-940
8080/8085 Floating Point Arithmetic Library User's Manual	98-452

Part No.	Title	Part No.
MCS-86 User's Manual	98-722	
MCS-86 Product Description (98-723)	205880	
AR 74 — Get Minicomputer Features at 10 x Speed with 8086 (98-921)	207310	
AR 82 — CPU Brings 6-Bit Performance (98-957)	207320	
AP 50 — Debug Strategies for 8089	207755	
AP 51 — Design 8086/8088/8089 with 8289	207760	
MCS-86 Assembly Macro Language Reference Manual	98-640	
MCS-86 Assembly Language Reference Guide (98-749)	205900	
Peripheral Design Handbook	98-676	
Peripherals Product Description	205600	
Microcomputers and Peripherals Pocket Guide (98-843)	205615	
AR 53 — Micro Interfacing Characteristics (8253) — (98-647)	207305	
AR 89 — Powerful I/O Processor Unloads CPU (8089)	207330	
AP 15 — 8255 Programmable Peripheral Interface (98-333)	207700	
AP 16 — Using the 8251 (98-334)	207705	
AP 31 — Using the 8259 (98-658)	207720	
AP 32 — 8275 and 8279 (98-576)	207725	
AP 35 — Crystals Specifications (98-652)	207730	
AP 45 — Using the 8202 Dynamic RAM Controller (98-809)	207745	
AP 48 — Direct Memory Access w/8257 DMA Controller	207750	
AP 54 — Dot Matrix Printer Controller Using the 8295 (98-816)	207765	
AP 59 — Using 8259A Programmable Interrupt Controller	207770	
INDUSTRIAL GRADE PRODUCTS		
Industrial Environment Brochure	206000	
Industrial Grade Product Book	206005	
MILITARY COMPONENTS		
Military Products Data Catalog	004150	
GENERAL DATA CATALOGS		
1979 Components Data Catalog	010200	
1979 Systems Data Catalog	506000	
PROTOTYPE MICROCOMPUTER KITS		
SDK-85 User's Manual	98-451	
SDK-86 Assembly Manual	98-697	
SDK-86 User's Guide	98-698	
ICS INDUSTRIAL CONTROL SERIES		
ICS 920 Digital Signal Hardware Reference Manual	98-801	
ICS 80 Industrial System Site Planning Guide	98-798	
ICS 80 Industrial Chassis Hardware Reference Manual	98-799	
ISBC 711 Analog Input Board Reference Manual	98-485	
ISBC 724 Analog Output Board Reference Manual	98-486	
ISBC 732 Combination Analog Input/Output Board Hardware Reference Manual	98-487	
ISBC 941 Industrial Digital Processor User's Guide	98-3077	
ICS Product Description (881-02)	500115	
ICS Brochure	500110	
AP 52 — Intel's Industrial Control Series in Control Applications (98-932)	511040	
SYSTEMS SOFTWARE		
RMX/80 User's Guide	98-522	
AP 33 — RMX/80 (98-577)	511020	
AP 47 — Using FORTRAN-80 for ISBC Applications (98-836)	452015	
OEM MICROCOMPUTER SYSTEMS		
ISBC 80/04 Hardware Reference Manual	98-482	
ISBC 80/05 Hardware Reference Manual	98-483	
ISBC 80/10 and ISBC 80/10A Hardware Reference Manual	98-230	

Part No.	Title	Part No.
AP 26 — ISBC 80/10-System 80/10		511000
RR 17 — ISBC 80/10 Reliability		509000
ISBC 80/20 and ISBC 80/20A Hardware Reference Manual		98-317
AR 28 — Control Engineering ISBC 80/20 Description		510100
ISBC 80/30 Hardware Reference Manual		98-611
AR 65 — Triple Bus Architecture (ISBC 80/30)		510140
ISBC 957 Inteltec ISBC 86/12 User's Guide		510140
AP 43 — Using the ISBC 957 (98-816)		511030
ISBC 86/12 Hardware Reference Manual		98-3075
AR 72 — 16-Bit Single Board Computer		510160
AR 69 — Dual-Port RAM Hikes Throughput (ISBC 80/30)		510150
ISBC 016 16K RAM Expansion Board Hardware Reference Manual		98-279
ISBC 032/048/064 Random Access Memory Boards Hardware Reference Manual		98-488
ISBC 094 4K-Byte CMOS RAM/Battery Backup Board Hardware Reference Manual		98-449
ISBC 104/108/116 Combination Memory and I/O Expansion Boards Hardware Reference Manual		98-277
ISBC 202 Double Density Diskette Controller Hardware Reference Manual		98-420
ISBC 204 Flexible Disk Hardware Reference Manual		98-568
ISBC 206 Disk Controller Hardware Reference Manual		98-567
ISBC 310 High-Speed Mathematics Unit Hardware Reference Manual		98-410
ISBC 416 16K PROM/ROM Expansion Board Hardware Reference Manual		98-265
ISBC 464 PROM/ROM Board Hardware Reference Manual		98-643
ISBC 501 Direct Memory Access Controller Hardware Reference Manual		98-294
ISBC 508 I/O Expansion Board Hardware Reference Manual		98-278
ISBC 517 Combination I/O Expansion Board Hardware Reference Manual		98-388
ISBC 519 Programmable I/O Expansion Board Hardware Reference Manual		98-385
ISBC 534 Four-Port Communications Expansion Board Hardware Reference Manual		98-450
ISBC 544 Intelligent Communications Controller Board Hardware Reference Manual		98-616
ISBC 556 Optically Isolated Programmable I/O Board Hardware Reference Manual		98-489
ISBC 569 Intelligent Digital Controller Hardware Reference Manual		98-845
ISBC 604/614 Cardcage Hardware Reference Manual		98-708
ISBC 635 Power Supply User's Manual		98-298
ISBC 640 Power Supply Hardware Reference Manual		98-803
ISBC 660 System Chassis Hardware Reference Manual		98-505
ISBC 915 GO-NO-GO Diskette Diagnostic and Monitor Program User's Manual		98-350
System 80/10 Microcomputer Hardware Reference Manual		98-316
System 80/20-4 Microcomputer Hardware Reference Manual		98-484
System 80/30 User's Guide		98-710
AR 48 — Reduce your Micro-based system design time		510110
AR 55 — Design Motivations for Multiple Processor Micro Systems		510120
AR 64 — Microcomputers — Single Chip or Single Board		510130
AP 28A — MULTIBUS Interfacing (98-587)		511010
Intel Delivers 8-bit/16-bit BM Configuration Envelopes		501100
INTELLEC MICROCOMPUTER DEVELOPMENT SYSTEM		
Inteltec 800 Operator's Manual		98-129
Inteltec Reference Manual		98-132
Inteltec Diagnostic Confidence Test Operator's Manual		98-386
Inteltec Double Density DOS Hardware Reference Manual		98-422
ISIS I DOS Operator's Manual		98-206
Diskette Operating System Manual		98-212
Paper Tape Reader Guide		98-016
Series II Hardware Reference Manual		98-556
Series II Model 210 User's Guide		98-557
Inteltec Series II Functional Description and Specifications (98-606)		404010
Inteltec Series II Installation and Service Manual		98-559
Inteltec Series Hardware Interface Manual		98-555
Success Manual for Single-Chip Microcomputer Users		402050
Success Manual for 8086 Users		402100
Microcomputer Development Package Booklet		404000
AR 97 — Minimizing Risk Through Use of Micro Development Systems		451130

	Title	Part No.
SOFTWARE		
iCIS Cobol Language Reference Manual		98-927
ICIS Cobol Packet Reference Card (98-929)		409100
2920 Assembly Language Manual		98-987
2920 Simulator User's Guide		98-988
FORTRAN-80 Programming Manual		98-481
FORTRAN-80 Reference Card (98-547)		400600
AR 73 — 8080 gets a "full blown" FORTRAN (98-844)		451125
PL/M Programming Manual		98-268
AR 59 — Modular Programming in PL/M		451115
PL/M 86 Programming Manual		98-466
BASIC-80 Reference Manual		98-758
BASIC-80 Reference Guide (98-774)		400705
AR 61 — Microprocessor Software Development Tools		451120
AR 98 — Software Development Package for 8086 System Designers		451135
ISIS II FORTRAN-80 Compiler Operator's Manual		98-480
ISIS II PL/M Compiler Operator's Manual		98-300
ISIS II PL/M 86 Compiler Operator's Manual		98-478
ISIS II 8085 Macro Assembler Operator's Manual		98-292
ISIS II System User's Guide		98-306
ISIS II Reference Card (98-841)		403350
ISIS II CREDIT User's Guide		98-902
CREDIT CRT-Based Text Editor Pocket Reference (98-903)		407700
MCS-86 Assembly Language Converter Operating Instructions for ISIS II Users		98-642
MCS-86 Assembly Operation Instructions for ISIS II Users		98-641
MCS-86 Software Development Utilities Operating Instructions for ISIS II Users		98-639
ICE-86 Operating Instructions for ISIS II Users		98-714
ICE-49 Operating Instructions for ISIS II Users		98-632
Multi-ICE Operating Instructions for ISIS II Users		98-672
iCIS-COBOL Compiler Operator's Instructions for ISIS II Users		98-928
8089 Assembler User's Manual		98-938
EMULATORS		
ICE-30 Reference Manual		98-220
ICE-41 Operator's Manual		98-465
ICE-41 Reference Card (98-766)		305075
ICE-48 Operator's Manual		98-464
MCS-48 ICE Reference Card (98-653)		303925
ICE-80 Reference Manual		98-167
ICE-80 Operator's Manual		98-185
ICE-85 Operating Instructions		98-463
ICE-85 Brochure		406215
ICE-86 Pocket Reference (98-838)		406310
Multi-ICE Reference Card (98-810)		406505
PROM PROGRAMMERS		
Universal PROM Programmer User's Manual		98-819
Universal PROM Programmer Reference Manual		98-133
PROMPT		
PROMPT 48 Microcomputer User's Manual		98-402
PROMPT 48 Reference Card (98-404)		304850
PROMPT 80/85 User's Manual		98-307
μSCOPE		
μScope 820 Operator's Handbook		98-526
μScope Reference Card (98-582)		408150
μScope 8080A Probe Service Manual		98-592
μScope 8085 Probe Service Manual		98-728
μScope Console Service Manual		98-593

ADD-IN/ADD-ON MEMORY SYSTEMS

in-7000/in-7001 Product Description
in-1670 Product Description
in-4011 Product Description
in-5034 Product Description
Series 90 Product Description — CM90
Series 90 Product Description — CM92
Series 90 Configuration Guide
AP 63 — Control and Interleaving BXP Standard Memory Bus

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

98-826

